Maya Dynamics

Alias|Wavefront
a Silicon Graphics Company
AP-M-DYN-01b

Maya Dynamics

Education Publishing Group:

Bob Gundu, Robert Magee, Deion Green

Global Education:

Lee Graft, John Patton, Shawn Dunn, Cory Mogk

Special Thanks to:

Ken Ibrahim, Jason Schleifer, Maya R+D team

Alias | wavefront

A SiliconGraphics Company

Introduction

Maya Dynamics

Dynamically derived animation is a large subject. This course of study introduces and explores the various segments that encompass this corner of the Maya universe.

What is Maya Dynamics?

Dynamics in Maya uses rules of physics to let you simulate natural forces in your animation. This animation is typically created by placing objects in a scene that then react to the forces applied to them. By creating an environment of fields, expressions, goals etc... the animator has artistic control over the affected objects; balancing the need for realism and animation requirements. Maya Dynamics is: The animation of rigid bodies, soft bodies, and particles, the use of dynamic constraints and the rendering strategies for hardware and software particle types.

After successfully completing this course, you will be able to:

- Utilize the Maya Dynamics tool-set within the Maya environment.
- Work with rigid bodies and dynamic constraints

- Control particles with fields, ramps, and expressions
- Render particles with both Hardware and Software methods
- Instance geometry with particle motion
- Optimize and troubleshoot dynamics scenes
- Dynamically animate NURBs and polygonal surfaces using soft bodies.
- Utilize Artisan functionality in conjunction with Maya Dynamics tools.

COURSE STRUCTURE

This course is taught in a lecture / lab format. The instructor will typically introduce the subject matter and demonstrate the workflow required for successful use of the Maya tool being presented. Pre-made files are provided for lab exercises where necessary. In lessons where involved concepts are introduced, simple test beds are explored with and without example scene file data.

Movie files are also provided as an aid in demonstrating the intended final outcome.

Who is this class designed for

This course is intended for the effects artist or technical director who has experience with Maya or another dynamics package. The researcher or scientist using Maya to visualize or model physical phenomena will also find value and insight from this course beyond what can be obtained from the Maya documentation and self experimentation.

Although knowledge of MEL and basic expression creation is not required, it is recommended and will allow for maximum participation of each student.

Rigid Bodies

The rigid body system in Maya allows for the animation of geometric objects in a dynamic system.

- **Active and Passive Rigid Bodies**—Active and passive rigid bodies are created to collide and react with one another in a realistic manner. Active objects typically fall, move, spin and collide with passive objects.
- **Rigid Body Constraints**—Rigid body constraints allow dynamic objects to be constrained or constrain each other. Spring, hinge, pin etc... are some of the constraint types.

Particles

Particles are objects that have no size or volume, they are reference points which are displayed, selected, animated and rendered differently then other objects in Maya.

- **Particle Object and Array attributes**—Like other nodes in Maya particles can be thought of as an object with a collective transform. They also contain attributes that control the individual particles using array attributes. The individual particle behavior can be controlled with ramps, scripts, and expressions.
- **Particle Expressions**—Particle expressions are a powerful and almost limitless method of animating particle parameters. Particle expressions share the MEL syntax and methodology. Functions such as linstep, sin, and rand() provide mathematical control over particle appearance and motion.
- **Particle Collisions**—Particle collision events provide a method for creating and killing particles when they collide with geometry. Particle collision event procedures can be used to trigger specific MEL scripted commands at collision time.
- **Emit function**—The emit function allows the user to create and position particles based on information directly derived from MEL and expressions. It requires an ample amount of MEL knowledge as more complicated usage of the emit function can get MEL-intensive.

Clip Effects

Clip effects provide the user with powerful and flexible tools for creating common dynamic effects. These are typically MEL derived scripts and expressions that automate the setup of the effect for the user. They also provide an excellent set of MEL and expression examples.

- **Flow**—The flow clip effect allows the user to select a curve as a motion path for particles.
- **Fire**—This clip effect will light an object on fire for both hardware and software rendering.
- **Smoke**—The smoke clip effect makes use of hardware sprites and is a good example of how to setup hardware sprites.

Particle Instancing

With particle instancing you can use particles to control the position and motion of instanced geometry.

- **Animated Instance**—Particle instancing is only part of the functionality. You can also control how the animated instance will behave on a per particle basis.
- **Cycled Instance**—With the particle instancer you can cycle through a sequence of objects to create the instanced motion.
- **Software Sprites**—The particle instancer also provides aim control of the instanced object. If this instanced object is a textured plane then it can be aimed at the camera creating a software renderable sprite method.

Goals

Goals are very powerful method of animating particles. Particles can have multiple goal objects and per particle attributes designed specifically for goal based interaction.

- **Goal weight and smoothness**—Goal weight and smoothness can be animated to provide particle movement that would be otherwise difficult to create with fields or expressions.
- **Per Particle Goal Attributes**—Goal attributes such as parentU and goalPP provide individual particle control. With the use of the parentId attribute these values can be transferred from one particle to another.

Particle Rendering

So what good is all this particle animation if you cannot render it out to contribute to the final shot?

- **Hardware Rendering**—Hardware rendering of particles provides a quick method of image creation. Typically these images are then taken to the compositor who sweetens and integrates them with the rest of the scene elements.
- **Software Rendering**—Software rendering allows for scene integration of particles and rendered objects. Volumetric particle rendering is also created with software rendered particle types. Shadowing and other lighting effects are also combined.
- **Compositing**—Without compositing much of this process would not be possible. Dynamics should always be viewed as another contributor to the elements that will make up the final image.

Softbodies

Animating geometry for fluid-like motion or with dynamic response is accomplished with softbody particles.

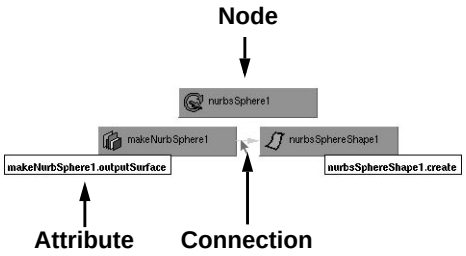
- **Softbodies**—Softbodies are geometry that have particles controlling the position and movement of the CVs or poly vertices of the objects.
- **Goals**—Goals are a fundamental part of controlling soft bodies. The goal weight controls the deviation of the soft body from its goal object
- **Springs**—Although springs can be applied to any particle, and even to dynamic geometry, they are especially suited to binding soft body components together.

THE DEPENDENCY GRAPH

While creating dynamics, it is a good idea to have a basic knowledge of how Maya's system architecture works. Maya's system architecture uses a procedural paradigm that lets you integrate traditional keyframe animation, inverse kinematics, dynamics and scripting on top of a node-

based architecture that is called the **Dependency graph**. If you wanted to reduce this graph to its bare essentials, you could describe it as *nodes with attributes that are connected*. This node-based architecture gives Maya its flexible procedural qualities.

Below is a diagram showing a primitive sphere's Dependency graph as shown in the Hypergraph view. A procedural input node defines the shape of the sphere by connecting attributes on each node.

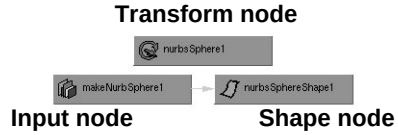


The Dependency graph

Nodes

Every element in Maya, whether it is a curve, surface, deformer, light, texture, expression, modeling operation or animation curve, is described by either a single node or a series of connected nodes.

A *node* is a generic object type in Maya. Different nodes are designed with specific attributes so that the node can accomplish a specific task. Nodes define all object types in Maya including geometry, shading, and lighting. Shown below are three typical node types as they appear on a primitive sphere.



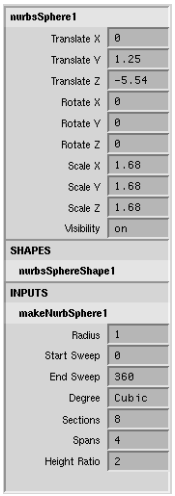
Node types on a sphere

Transform node - Transform nodes contain positioning information for your objects. When you move, rotate or scale, this is the node you are affecting.

Input node - The input node represents options that drive the creation of your sphere's shape such as radius or endsweep.

Shape node - The shape node contains all the component information that represents the actual look of the sphere.

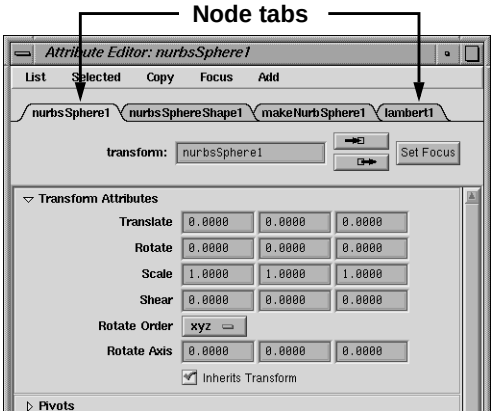
Maya's user interface presents these nodes to you in many ways. Below is an image of the Channel box where you can edit and animate node attributes.



Channel box

Attributes

Each node is defined by a series of attributes that relate to what the node is designed to accomplish. In the case of a transform node, *X Translate* is an attribute. In the case of a shader node, *Color Red* is an attribute. It is possible for you to assign values to the attributes. You can work with attributes in a number of user-interface windows including the *Attribute Editor*, the *Channel box* and the *Spread Sheet Editor*.



The Attribute Editor

One important feature in Maya is that you can animate virtually every attribute on any node. This helps give Maya its animation power. You should note that attributes are also referred to as *channels.*

Connections

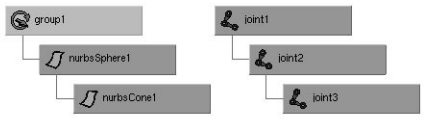
Nodes don't exist in isolation. A finished animation results when you begin making connections between attributes on different nodes. These connections are also known as *dependencies.* In the case of modeling, these connections are sometimes referred to as *construction history*.

Most of these connections are created automatically by the Maya user-interface as a result of using commands or tools. If you desire, you can also build and edit these connections explicitly using the *Connection editor*, by entering *MEL* (Maya Embedded Language) commands, or by writing MEL-based expressions.

Hierarchies

When you are building scenes in Maya, you can build dependency connections to link node attributes. When working with transform nodes or joint nodes, you can also build hierarchies which create a different kind of relationship between your objects.

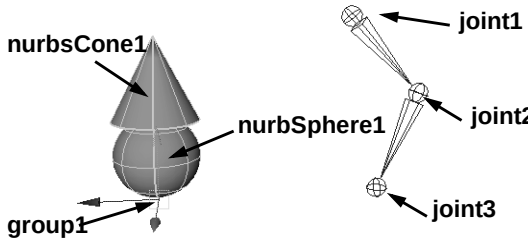
In a hierarchy, one transform node is *parented* to another. When Maya works with these nodes, Maya looks first at the top node, or *root* node, then down the hierarchy. Therefore, motion from the upper nodes is transferred down into the lower nodes. In the diagram below, if the *group1* node is rotated, then the two lower nodes will rotate with it. If the *nurbsCone* node is rotated, the upper nodes are not affected.



Object and joint hierarchy nodes

Joint hierarchies are used when you are building characters. When you create joints, the joint pivots act as limb joints while bones are drawn between them to help visualize the joint chain. By default, these hierarchies work just like object hierarchies. Rotating one node rotates all of the lower nodes at the same time.

When you are working with characters, you will use *inverse kinematics* to reverse the flow of the hierarchy.



Object and joint hierarchies

The Hypergraph

In Maya, you can visualize hierarchies and dependencies using the hypergraph. The following steps demonstrates how to work with various node types in the Hypergraph.

Working with hierarchies and dependencies

If you understand the idea of *nodes with attributes that are connected*, then you will understand the Dependency graph. You can see what this means in Maya by building a simple primitive sphere.

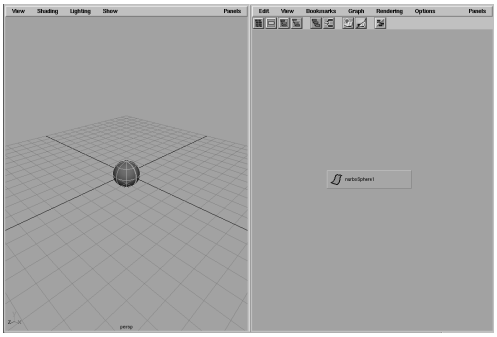
1 Set up your view panels

To view nodes and connections in a diagram format, the Hypergraph panel is required along with a Perspective view.

- Select **Panels** → ***Layouts*** → ***2 Side by Side***.
- Set up a Perspective view in the first panel and a Hypergraph view in the second panel.
- Dolly into the Perspective view to get closer to the grid.

2 Create a primitive sphere

- Go to the modeling menu set.
- Select **Primitives** → ***Create NURBS*** → ***Sphere***.
- Press **5** to turn on smooth shading and **3** to increase the surface smoothness of the sphere.



New sphere

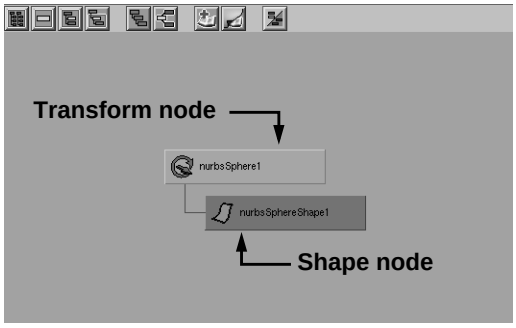
3 View the shape node

In the Hypergraph panel, you are currently looking at the scene view. The scene view is focused on *transform nodes*. This node type lets you set the position and orientation of your objects.

Right now, only a *nurbsSphere* node is visible. In actual fact, there are two nodes in this hierarchy but the second is hidden by default. At the bottom of most hierarchies, you will find a *shape node* which contains the information about the object itself.

- In the Hypergraph panel, select **Options** → ***Show*** → ***Shape nodes***.

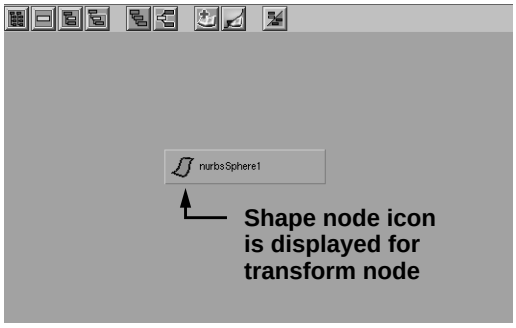
  You can now see the *transform node* which is, in effect, the positioning node and the *shape node* which contains information about the actual surface of the sphere. The transform node defines the position of the shape below it.



Transform and shape nodes

- In the Hypergraph panel, select **Options** → ***Show*** → ***Shape nodes*** to turn these off.

  You will notice that when these nodes are expanded, the shape node and the transform node have different icons. When collapsed, the transform node takes on the shape node's icon to help you understand what it going on underneath.

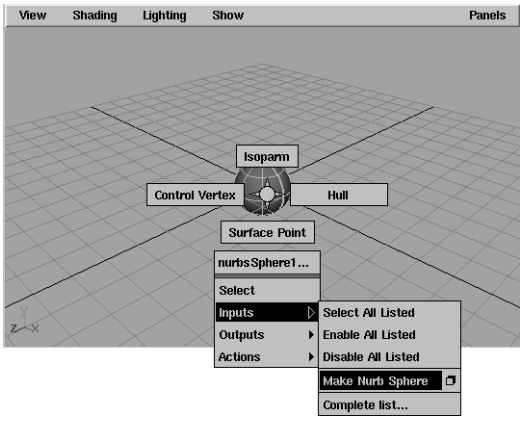


Transform node on its own

4 View the dependencies

To view the dependencies that exist with a primitive sphere, you need to take a look at the upstream and downstream connections.

- Click on the sphere with the right mouse button and select **Inputs** → ***Make Nurb Sphere*** from the marking menu.



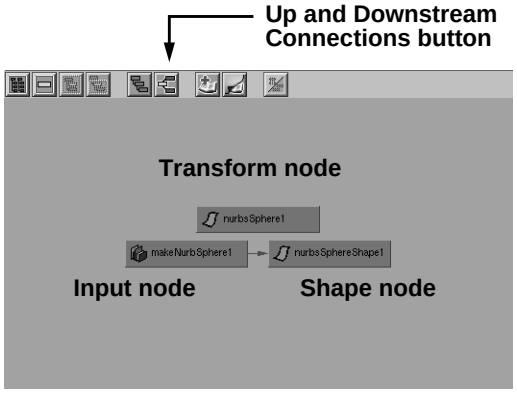
Selection marking menu

Note: You can also select the input node by choosing it in the Channel box.

- In the Hypergraph panel, click on the **Up and Downstream Connections** button.

  See the original transform node which is now separated from the shape node. While the transform node has a hierarchical relationship to the shape node, their attributes are not dependent on each other.

  The *input node* called *makeNurbSphere* is a result of the original creation of the sphere. The options set in the sphere tool's option window, have been placed into a node that feeds into the shape node. The shape node is dependent on the input node. If you change values in the input node, then the shape of the sphere changes.



Sphere dependencies

5 Edit the attributes in the Channel box

In the Channel box, you can edit attributes belonging to all of the node types. This lets you affect both hierarchical relationships and dependencies.

If you edit an attribute belonging to the *makeNurbSphere* node, then the shape of the sphere will be affected. If you change an attribute belonging to the *nurbSphere* transform node, then the positioning will be changed. Using the Channel box will help you work with the nodes.

- For the transform node, change the **Rotate Y** value to **45**.
- For the input node, change the **Radius** to **3**.

  You can set attribute values to affect either the scene hierarchy or the Dependency graph.

Animating the sphere

When you animate in Maya, you are changing the value of an attribute over time. Using keys, you set these values at important points in time, then use tangent properties to determine how the attribute value changes between the keys.

The key and tangent information is placed in a separate animation curve node that is then connected to the animated attribute.

1 Select the sphere

- In the Hypergraph panel, click on the **Scene Hierarchy** button.
- ***Select*** the *nurbsSphere* transform node.

2 Return the sphere to the origin

Since you earlier moved the sphere along the three axes, it's a good time to set it back to the origin.

- In the Channel box, change the **Rotate Y** attribute to **0**.

3 Animate the sphere's rotation

- In the Time slider, set the playback range to **120** frames.
- In the Time slider, go to frame **1**.
- Click on the **Rotate Y** channel name in the Channel box.
- Click with your right mouse button and select ***Key Selected*** from the pop-up menu.

  This sets a key at the chosen time.
- In the Time slider, go to frame **120**.
- In the Channel box, change the **Rotate Y** attribute to **720**.
- Click with your right mouse button and select ***Key selected*** from the pop-up menu.
- Playback the results.

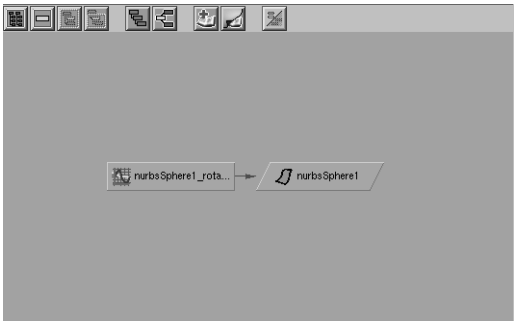
  The sphere is now spinning.

4 View the Hypergraph dependencies

- In the Hypergraph panel, click on the **Up and Downstream Connections** button.

You see that an animation curve node has been created and connected to the transform node. The transform node is now shown as a trapezoid to indicate that it is connected to the animation curve node. If you click on the connection arrow, you will see that the connection is to *Rotate Y*.

If you select the animation curve node and open the Attribute Editor, you will see that each key has been recorded along with value, time and tangent information. You can actually edit this information here, or use the Graph Editor where you get more visual feedback.



Connected animation curve node

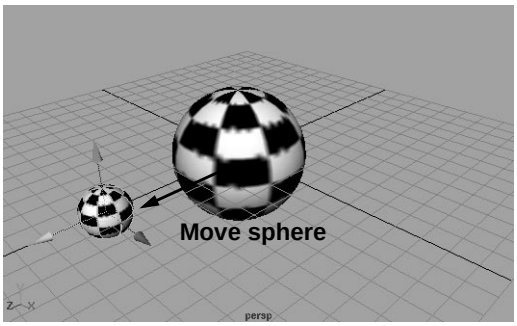
Parenting in the Hypergraph

So far, you have worked a lot with the dependency connections but not with the scene hierarchy. In a hierarchy, you always work with transform nodes. You can make one transform node the *parent* of another node, thereby creating a child which must follow the parent.

You will build a hierarchy of spheres that are rotating like planets around the sun. This example is a helpful way to understand how scene hierarchies work.

1 Create a new sphere

- In the Hypergraph panel, click on the **Scene Hierarchy** button.
- Go to the **Modeling** menu set.
- Select **Primitives** → ***Create NURBS*** → ***Sphere***.
- **Move** the sphere along the Z axis until it sits in front of the first sphere.
- Press **3** to increase the display smoothness of the sphere.
- Go to the **Rendering** menu set.
- Apply a checker shader to both spheres

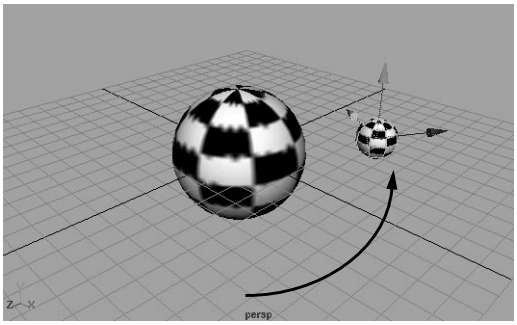


Second sphere

2 Parent the sphere to the first sphere

- In the Hypergraph, drag the node icon with **MMB** for the second sphere onto the first sphere. Now they are parented together.
- Playback the scene.

  The second sphere rotates along with the first sphere. It has inherited the motion of the original sphere.



Rotating hierarchy

Summary

You should now know the objectives and outline and how the Dependency Graph works in Maya.

Chapter 1

1 Rigid Body Dynamics

RIGID BODY FUNDAMENTALS

This lesson introduces the fundamental tools and techniques required to achieve realistic “solid” collisions in Maya using rigid bodies.

You will learn the following:

- Creating rigid bodies
- Differences between active and passive rigid bodies
- Creating and connecting fields to rigid bodies
- Working with the rigid body solver and its attributes
- Combining keyframing with rigid body dynamics

WHAT IS A RIGID BODY?

Rigid bodies within the context of Maya are defined as any object whose surface does not deform when a collision occurs. Common examples in nature would be billiard balls, floors and ceilings, or a bowling ball. Of course in the real world theses surfaces do actually deform to a very small extent when a collision occurs. Within Maya, we overlook this minimal deformation and consider an object as either rigid or not rigid to simplify the process.

In Maya, any NURBs or polygonal surface can contain rigid body properties. Curves, particles, and lattices for example cannot become rigid bodies since they contain no surface information. But surfaces that are made from curves, particles or lattices for example can be rigid bodies.

Soft Bodies are NURBs or polygonal surfaces that have particle like control of there respective CVs or poly Vertices. Soft bodies will be explored in later lessons.

Active vs. Passive rigid bodies

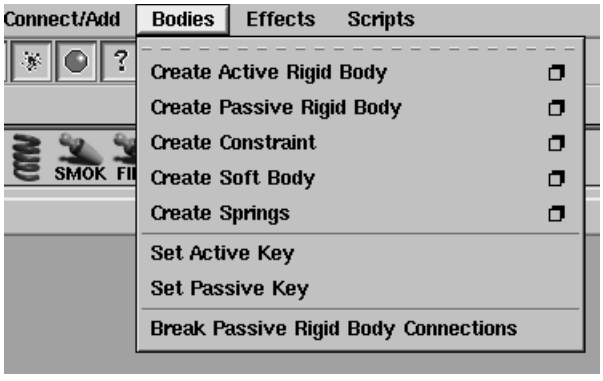
Maya's rigid bodies are divided into two categories: *active* and *passive*. There are important distinctions between these two types of rigid bodies.

	ACTIVE	PASSIVE
Keyframable	no	yes
Responds to collisions	yes	no
Causes Collisions	yes	yes
Affected by fields	yes	no

Active and Passive Rigid Bodies compared

The Bodies Menu

The **Bodies** menu is used to create rigid and soft bodies, create dynamic constraints, and keyframe the active and passive state of objects.



Important Rigid Body Nodes

The rigid body command you choose will create several new important nodes and attributes for each selected object. These nodes (and their associated attributes) can be viewed in the Channel Box, Hypergraph, Outliner, or Attribute Editor. In the channel box you will notice the following nodes are created for each selected object:

rigidBody

The *rigidBody* node is located under the **SHAPES** section for each selected rigid body object in the Channel Box.

To view and select a *rigidBody* node from the Outliner, you may first need to show shapes by selecting **Outliner** → **Display** → **Shapes**. The rigidBody node will then appear as a child of the object's transform node.

The attributes within this node contain information that determines the active / passive status of the rigid body and various controls relating to the properties of each specific rigid body object.

rigidSolver

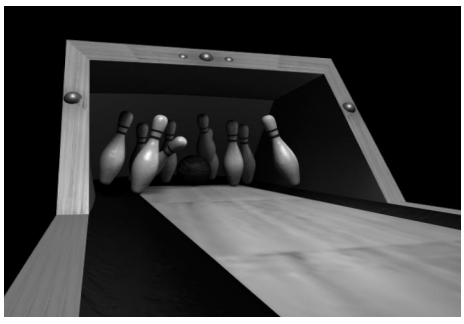
The *rigidSolver* node provides control over the evaluation of the rigid body dynamics. This node is listed under both the **INPUT** and **OUTPUT** section within Channel box for the selected item. By default, one *rigidSolver* node is used to control the evaluation of all rigid bodies in the scene.

time

The *time* node determines when the rigidSolver's evaluations will take place. This is useful if you wish to have multiple simulations within the same scene running based on different time parameters.

BOWLING ALLEY EXAMPLE

This example incorporates the use of active and passive rigid bodies and is intended to familiarize you with the process of setting up a rigid body simulation in Maya.



The bowling alley example

1 Load the file

- Select **File** →**Open Scene**.
- Select the file named *bowling.ma***.**

2 Create the active rigid bodies

- Use the Outliner to **select** *pin1* to *pin10.*
- **Control-click** *bowlingBall* to the selection.
- Select **Bodies** → **Create Active Rigid Body - ❐**
- Press **Reset**, to set the options to the default state.
- Press **Create**.

3 Create the passive rigid bodies

- Use the Outliner to select the following objects:

  bowlingLane, leftGutter, rightGutter, innerCage, backCage
- Select **Bodies** → **Create Passive Rigid Body - ❐**
- Press **Reset** to set the options to the default state.
- Press **Create**.

4 Set rigid body attributes for the bowling ball

- Select *bowlingBall***.**
- Locate the rigidBody node for this object in the **SHAPES** section of the Channel Box.
- Set the following attribute values for the rigidBody node:

  initialVelocityZ to **-55;**

  initialSpinX to **-1400;**

  mass to **400;**

  bounciness to **0;**

  damping to **0;**

  applyForcesAt to **verticiesOrCvs;**

  standIn to **sphere**

5 Set rigid body attributes for the bowling pins

- Select all of the bowling pins.
- Enter the following attribute values for the rigid body nodes of the selected objects.

  mass to **35**

  bounciness to **0.5**;

  damping to **0**;

  applyForcesAt to **verticiesOrCvs**;

  static friction to **.05**;

  dynamic friction to **.05**

We'll discuss the specific definition of these attributes in a later chapter, for now, we just want to get some collisions happening.

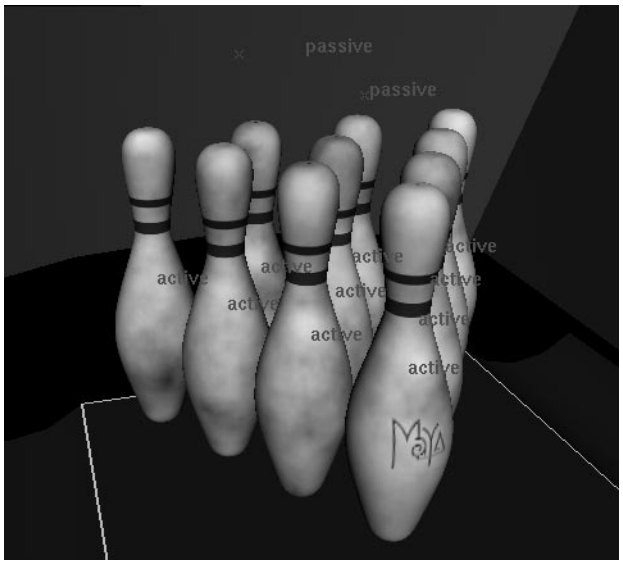
6 Turn on Dynamic Labels

In some cases when dealing with complex scenes containing large numbers of rigid bodies, it is difficult to keep track of which rigid bodies are passive and active. When set to **ON**, the **dynamicLabels** attribute will display a small label next to each rigid body in the viewport indicating if it is an active or passive rigid body.

Note: Other dynamic components such as dynamic constrains (discussed later) also have labels associated with them.

- Select any rigid body object in the scene.
- Click on the *rigidSolver* node near the bottom of the Channel Box.
- Set the **displayLabel** attribute to **on**.
- Rewind to display the dynamic labels.

  The labels are placed at the object's center of mass which is represented by a small X on each object.



Displaying rigid body labels at the object's center of mass

Note: When multiple items are selected, (denoted by**...** in the name field of the channel box) it is only necessary to enter the attribute values for the first selected object. The change will be made to all selected items that contain the attribute being edited.

---

7 Create a gravity node

- Click in the viewport to deselect any currently picked objects.

- From the Dynamics menu select **Fields** → **Create Gravity-** ❐✎
- Press **Reset** then **Create**.

8 Connect the objects to gravity

- Select all 10 pins and the bowlingBall.
- Select the gravity field last.
- Select **Connect/Add** → **Connect to Field.**

Tip: The same results could have been achieved by selecting the pins and ball then choosing **Fields** → **Create Gravity**. This process would automatically connect the selected items to the chosen field.

A word about selection

If you are doing your selection in the Hypergraph you will find that Shift selecting is required. If you are doing your selection in the Outliner then Cntrl selection will give you individual selection. Shift selection in the Outliner gives you selection of the objects that lie between the first selected and the second selection action. You can also **LMB** drag to select objects and attributes in various places in Maya.

Going forward, it is assumed that selection behavior and preferred selection technique is at your discretion. The Outliner and the HyperGraph have their inherent strengths for organization and object manipulation. You can choose to use the one you are most comfortable with for the task at hand.

9 Test the results

- Rewind then playback the scene

  The bowling ball is projected along the Z-axis while spinning on its local X-axis. The ball collides with the pins, the pins collide with each other and the barriers of the bowling alley. Notice that the barriers remain stationary.

Caching

Caching allows Maya to evaluate calculations once per frame and store the result of those calculations in memory (RAM) where they can be accessed during subsequent playback at much faster speeds.

When caching is enabled, Maya will continue to use the cached version of the data as the “master” until the cache has been deleted. For this reason, any modifications made to the simulation after caching the data will not exist in the currently cached version of the playback since those changes were not part of the original calculations that were computed during the cache run-up.

It is best to tweak values then cache the scene. Once you have evaluated those changes and wish to make more, delete the cache, make the new changes, then re-cache the scene again.

Note: The first playback cycle where Maya records the calculations into RAM is commonly referred to as a "run-up."

1 Cache the playback

- In the timeline, set the playback range to start at frame **1** and end at frame **100**.
- Select *bowlingBall*.
- Click on the **rigidSolver** node in the Channel Box and set the following:

  cacheData to **1**

  A value of **1** corresponds to **on**, a value of **0** corresponds to **off**.

Note: You can also enable or disable caching by selecting **Solvers** → **Scene Caching** → **Enable (or Disable).**

- Rewind then playback the scene.

  You can scrub in the timeline after the simulation has played through once.

2 Delete the cache

- Select **Solvers** → **Scene Caching** → **Delete** to clear the previously cached data from memory.

3 Modify the attribute values

- Experiment with the values used for **initialVelocityZ**, **initialSpinX**, **mass**, and **bounciness** to see how the simulation results differ.
- Re-cache the scene so you can see the result playback in real time.
- Repeat this process until you've achieved the desired motion.

Combining Keyframes with Dynamics

In some cases, relying solely on the influence of fields, initial velocity, initial spin, and other dynamic attributes to animate objects may not provide the level of control required. In these cases, it is useful to combine keyframing techniques with dynamic techniques to tune the motion of the animation.

1 Load the file

- Select **File**→**Open.**
- Select *keyedDynamics.ma*

When you playback the scene, you will notice the ball has an initial velocity in the direction of the negative Z axis which propels it into the pins as in the previous file.

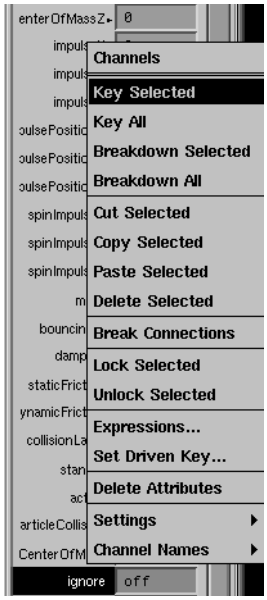
2 Keyframing the Ignore attribute

Since no substantial collisions occur in this animation until many frames into the simulation, Maya can ignore the computation of the pins until near the time of collision to speed up playback.

- Verify playback start range is at **1**.
- Select all 10 bowling pins
- In the Channel Box use **RMB** to keyframe the **Ignore** attribute as follows:

  Keyframe **Ignore** at frame **1** to **on**

  Keyframe **Ignore** at frame **50** to **off**



Using the MMB in the Channel Box to keyframe

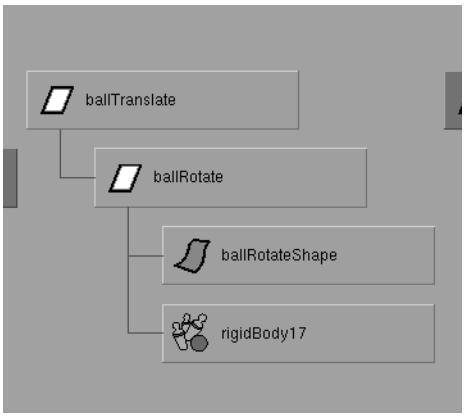
3 Group the bowlingBall

Creating a group on the bowlingBall provides a second transform with which to apply hierarchical animation. In this case a motion path.

- Select *bowlingBall.*
- Select **Edit→Group**
- Rename the new group to *ballTranslate.*
- Rename *bowlingBall* to *ballRotate*.

  ballTranslate will handle the translation of the ball along the motion path. *ballRotate* is where you will keyframe rotation.

  The hierarchy of the ball should appear as shown below:



Hierarchy of bowling ball in the Hypergraph with Display Shapes turned ON

4 Center the Pivot on the ballTranslate

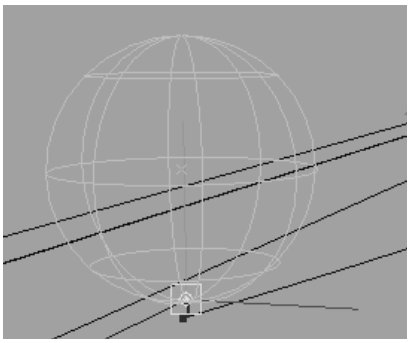
When you grouped the ball, the pivot point was placed at the origin.

- Select *ballTranslate*.
- Select **Modify** → **Center Pivot.**

5 Move ballTranslate's pivot to its base

It is important to move the pivot point of *ballTranslate* to the base of the bowling ball so it is attached to the motion path at the base instead of the center of the ball.

- Select *ballTranslate* and press **W** to enable the **move** tool.
- Press **Insert** on the keyboard to switch to pivot mode.
- Move the pivot point manipulator to the base of the ball.



ballTranslate pivot at base

6 Switch the bowling ball to a passive rigid body

In order for the ball to travel on the path, it needs to be a passive rigid body.

- In the Channel Box enter **off** for the **active** attribute.

7 Add the passive rigid body to the motion path

- Set the playback frame range from **1** to **60**.

- Rewind to frame **1**.
- **Select** *ballTranslate* then, **shift-select** *ballCurve*.
- Select **Animate** → **Paths** → **Attach to Path -** ❐, and set the following options:

  Time Range to **Time Slider;**

  Follow to **ON;**

  Front Axis to **Z;**

  Up Axis to **Y;**

  Up Direction to **World Up;**
- Press **Attach** to apply *ballTranslate* to the motion path.

8 Test the results

- Set the playback frame range to start at frame **1** and end at frame **200** then playback the scene.

  Notice that the ball translates along the motion path from frame 1 to 60. However, the ball does not continue past the end of the motion path and does not respond to the collisions from the pins.

  In order to achieve this behavior, it is necessary to keyframe the **passive** rigid body so it becomes an **active** rigid body when the ball approaches the end of the motion path.

9 Keyframe the active/passive state of the ball

In order for the ball to be both active or passive at different frames in the animation you will keyframe its **active** state attribute using special menus.

- Rewind to frame **1.**
- Locate the **rigid body** node for *ballRotate* in the Channel Box.
- Select **Bodies Set** → **Passive Key** to key this as a passive body
- Advance to frame **60.**
- Select **Bodies Set** → **Active Key** to key this as an active body

---

Note: Keying the active/passive state using these menus is the most reliable method. However, be aware that this does create keyframes on all objects in the hierarchy. Sometimes setting the active/passive keyframe a frame or two before the motion path ends helps to prevent problems with order of evaluation.

Keyframe the ball rotation

Put some spin on the ball by keyframing the rotation of the *ballRotate* node.

1 Keyframe ballRotate

- Go to frame **1;**
- Select *ballRotate* node.
- Set a keyframe on the **rotateX** channel of *ballRotate***.**

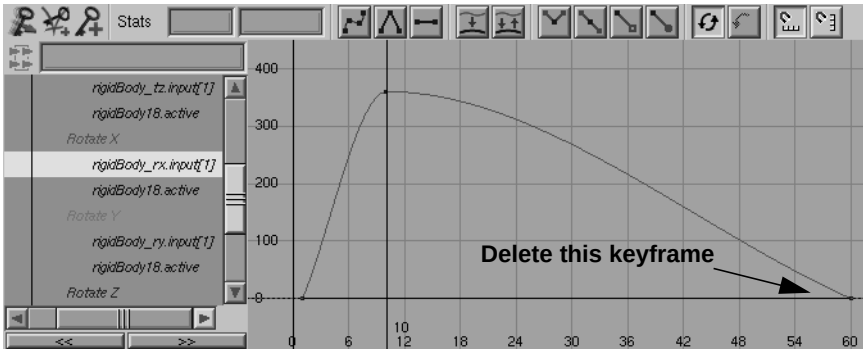
- Advance to frame **10**.
- Enter a value of **360** for the **rotateX** attribute of *ballRotate*.
- Set another keyframe for that attribute.

2 Cycle the rotateX animation curve

The cycling option in the Graph Editor causes the rotation that was just set up to repeat in 10 frame increments throughout the animation. Therefore, the same rotation motion that occurred on frames 1-10 will also occur on frames 11-20, 21-30, and so on.

- Open the Graph Editor by selecting **Window** → **Saved Layouts** → **Persp/Graph/Outliner**
- Locate and select the *rigidBody_rx.input[1]* curve which is within the **rotateX** curve hierarchy in the Graph Editor.

  Press **f** to *frame selected.*
- Delete the keyframe at frame 60 for this curve only.



- With the same curve still selected, choose **Curves** → **Post Infinity** → **Cycle with Offset**.
- Select **Toggle View** → **Infinity** to **ON**.

  This displays the cycled curve data in the Graph Editor. This option can be left on or toggled to **OFF** again to avoid clutter in the Graph Editor.

3 Test the scene

- Play back the animation.

  The ball translates and rotates along the motion path as a passive rigid body.It continues moving and rotating after the motion path animation has completed as an active rigid body. This continued motion is the combined result of the gravity field, motion inherited from the path animation, and keyframed rotation.

Exercises

Use what you've learned here to build a house of cards on a table. Apply fields to the cards to make them collide with each other and the table.

CONCLUSION

You now have a basic understanding of how Maya's rigid body dynamics system works. In upcoming chapters we will discuss the various rigid body attributes and solver attributes in greater detail so you understand their specific function better. The upcoming chapters will also focus more on tuning and optimizing the simulation. You should now be able to:

- Create Active and Passive Rigid bodies
- Use keyframes, motion paths, and fields to control rigid body motion
- Keyframe the active / passive state of a rigid body
- Use the Ignore attribute and caching to speed playback
- Use cycling in the Graph Editor
- Recognize and understand important rigid body nodes such as time, rigidSolver and rigidBody

Chapter 2

2 Rigid Body Constraints

This lesson focuses on working with Maya's rigid body constraints. The two examples will be a hanging mobile, and a medieval catapult.

In this lesson you will learn the following:

- Constraint types
- Animating constraint parameters
- Parenting constraints
- Application with dynamic and non-dynamic constraints
- Rigid body groups

CONSTRAINT TYPES

There are 5 constraint types divided into two categories:

Dual body constraints

The *dual body* constraints allow for constraining of a rigid body to a point in space or to another rigid body.

Pin

Hinge

Spring

Single body constraints

The *single body* constraints allow for the constraint of a rigid body to only a point in space or, in the case of the Barrier, to a plane in space.

Barrier

Nail

Constraint descriptions:

Pin constrains two active rigid bodies to each other, or an active and a passive rigid body to each other. It does not allow for constraint to a point in space. The pinning point or pivot is adjustable and can be keyframed on and off.

Nail constrains an active rigid body to a point in space. This point can be grouped and translated under another object as a child.

Barrier creates a boundary that an active rigid body cannot pass through.

Hinge constrains an active rigid body to another active or passive rigid body with a user defined pivot orientation. This pivot constrains the motion to one axis.It can also constrain a rigid body to a point in space.

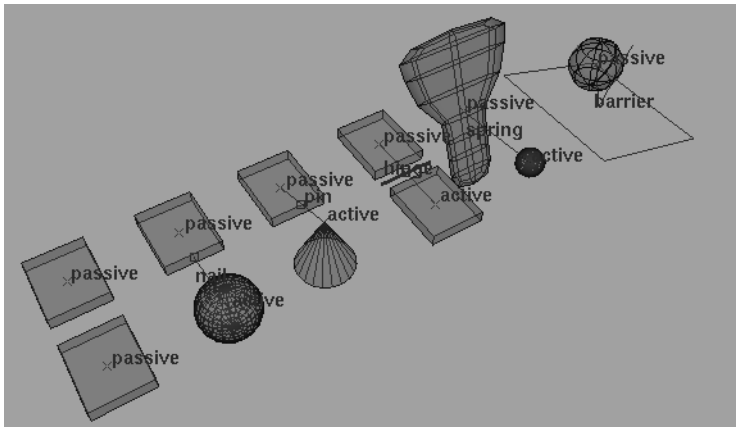
Spring constrains an active or a passive rigid body to another active rigid body or a point in space. The spring constraint contains attributes that control the elastic properties, **Stiffness, Restlength** and **Damping**.

Auto Creating

When you select an object to be dynamically constrained to a point or another object, Maya will automatically turn the necessary objects into rigid bodies if they are not rigid bodies already. This feature is controlled by the **Auto Create Rigid Body** flag in the **Dynamics** section of the **General Preferences.**

Examples

The scene file *constraint_Examples.mb* contains a sample application of each constraint type.



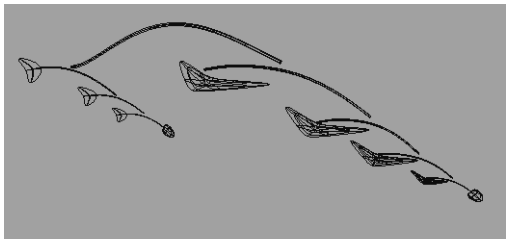
Constraint examples scene

Example: MayaMobile

In this exercise you will hook up the provided geometry into a free hanging kinetic sculpture.

1 Open a file

- Open the file *moblie_geometry.mb*



The mobile_geometry file

This file consists of a group node with the mobile pieces organized within.

The main structure is one of arms that hang on each other and objects that will hang under these arms. The Base object is a sphere that we can use as a reference point in space to hang the sculpture from.

2 Create a Nail constraint to support the entire structure

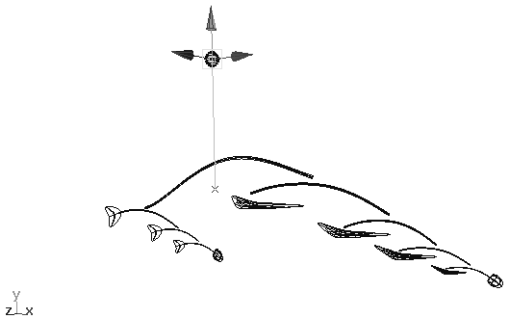
A nail constraint will allow you to hang the sculpture from a point in space.

- Select *mainBoom*.
- Select **Bodies** → **Constraints -** ❐, and set the following:

  Constraint Type to **Nail**
- Press **Create.**
- Translate the *rigidNailConstraint* pivot to the center of the *base* object.

- Parent the nail constraint to the *base* object so that you can move the entire sculpture by moving the *base* object.

  The nail constraint constrains the center of mass of the mainBoom to the nail constraint pivot point.



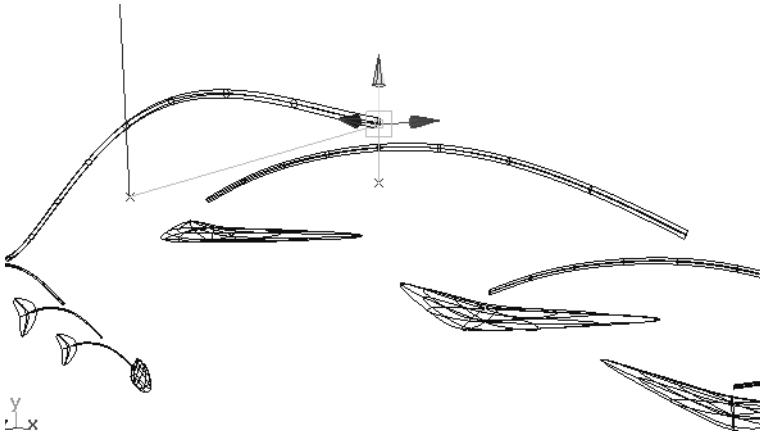
Move the nail constraint point to the base objects position

3 Create Pin constraints for the arms of the sculpture

Pin constraints are a good choice for this type of work.

- Select the *mainBoom* then *armPlane1*.
- Select **Bodies** → **Constraints** - ❐, and set the following:

  Constraint Type to **Pin;**
- Press **Create**
- Translate *rigidPinConstraint* pivot to the end of the *mainBoom*.
- Repeat this process for the other booms.



Move the pin constraint pivot to the hanging point

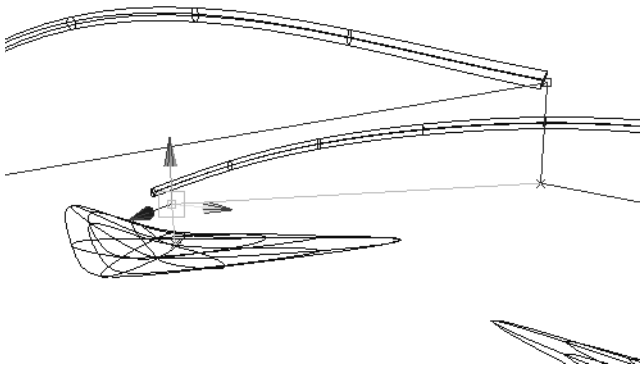
4 Create Pin constraints for the sculpture pieces

In the same manner that you Pin constrained the booms, you will apply Pin constraints to the other pieces, moving the Pin Pivot to the appropriate location.

You will notice that the constraint is created between the two object's center of mass.

Once the constraint is created you can move the constraint pivot to the desired location.

Tip: HINT: by moving the constraint pivot point to vertically above the hanging objects center of mass you will avoid excessive settling of the mobile at the start of the simulation.



Tip: You can use the **g** hotkey to repeat the last command. In this example it will Apply Pin Constraint.

5 Create Gravity on the sculpture components

- Select all of the constrained objects using **Cntrl-shift** selection methods.
- Select **Fields** → **Create Gravity.**
- Test with playback to confirm they all fall under the influence of gravity.

6 Adjust mass and center of mass to balance the sculpture

The attributes of mass and center of mass will influence the balance of the sculpture. The sculpture lies on the XY plane. You should restrict your positioning of center of mass using the **centerOfMassX** and **centerOfMassY** attributes.

There is no manipulator for center of mass.

- Select the **centerOfMassX** attribute in the Channel Box.
- **MMB-drag** in the viewport to interactively change the value.

  Typically you can balance the leaf pieces by moving the center of mass of X

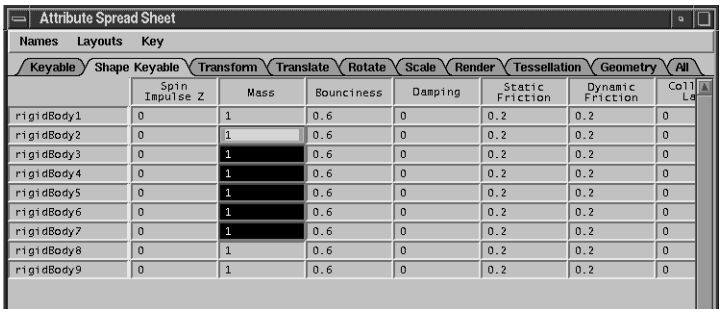
  The plane and arm pieces react well to manipulating the mass.

  The plane objects will also need their center of mass in Z adjusted so that they maintain straight and level flight.

7 Adjust the mass of the sculpture pieces with the Spreadsheet

Keep in mind that the mass of the booms is taken into account as well.

- Select the rigid body components of the sculpture including the booms by selecting **Edit → Select All by Type → Rigid Bodies.**
- Select **Window → General Editors → Attribute Spread Sheet...**
- Scroll down in the Spreadsheet to locate the rigid body nodes.
- Locate the mass attributes of the selected objects and adjust as necessary.



Attribute Spread Sheet

Tip: You can drag select multiple cells in a single or multiple columns to make simultaneous value changes. You can **cntl**-**shift** select cells. Entire rows or columns can be selected by clicking on the row or column title.

8 Add initial spin and fields to animate the piece

This is where things get real tricky. It is one thing to get it hanging good but then to animate it we see that objects that are constrained will interact with the constraining objects. Experiment with Spin Impulse, Impulse and Initial Velocity.

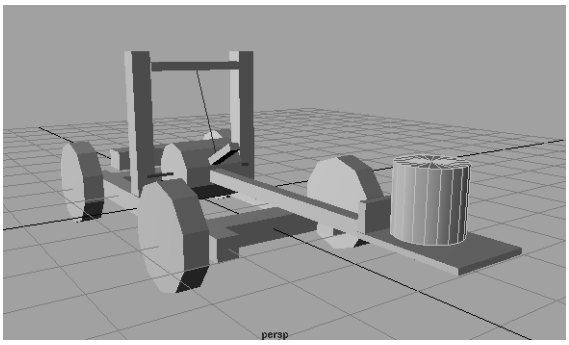
Initial Spin on the booms and leafs works well.

Impulse can be applied to the objects but remember that an impulse is applied on every frame so that it can create some extreme accelerations.

This can be a difficult exercise. Do not get discouraged if you can not quickly balance the sculpture.

Example: Medieval Destruction

In this exercise you will construct a Catapult using the hinge and spring constraints.



Catapult with spring and hinge constraint

1 Open scene file

- Open the file *catapult.mb*

  This file contains the catapult geometry. Note the following inventory of objects:

  overhead - Static anchor for spring constraint

  lever - Armature anchor for spring constraint

  frame - Geometry of catapult frame

  wheels - Geometry of catapult wheels

  boom - Armature arm

  boomWheel - Armature fulcrum

2 Spring Constrain the lever to the overhead

Use the spring constraint to pull the armature towards the overhead anchor which will remain stationary.

- **Shift-select** *lever* and *overhead* objects.
- Select **Bodies** → **Constraints** - ❒, and set the following:

  Constraint Type to **Spring**
- Press **Create.**

  This converts lever and overhead to active rigid bodies and builds a spring constraint between them.

3 Change the overhead object to a passive rigid body

- Select the *overhead* object.
- In the Channel Box for the **active** attribute enter **0** or **off**.

4 Orient constrain the boomWheel to the lever

The lever object will rotate the *boomWheel* object so you must orient constrain the *boomWheel* to the *lever*.

- Select the *lever* then the *boomWheel* object.
- Select **Constrain** → **Orient.**

  This will create a *boomWheel_orientConstraint* under the *boomWheel* object.

Note: Be careful not to confuse the animation constraints with the dynamic rigid body constraints.

5 Hinge constrain the lever object

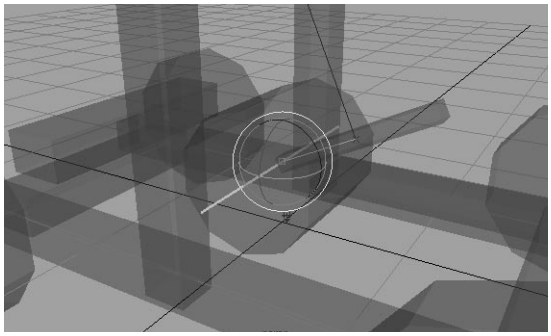
To get the lever to only rotate about the same axis as the *boomWheel* you will use a hinge constraint on the lever and then move the hinge to the center of the boomWheel.

- Select the *lever* object.
- Select **Bodies** → **Constraints** - ❐, and set the following:

  Constraint Type to **Hinge**.
- Press **Create**.

6 Reposition the hinge

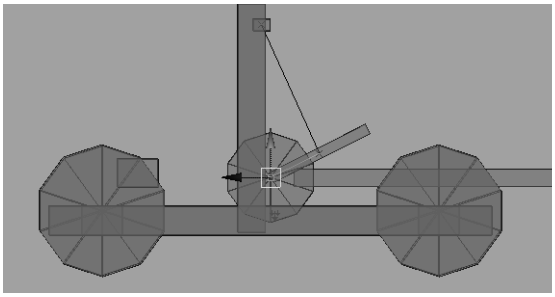
- Select the hinge object and rotate it so that it is parallel with the x-axis of the *boomWheel*.



Hinge constraint orientation

- Translate the hinge to the center of the boomWheel.

  You should use an orthographic view to align this accurately.



Hinge constraint position

7 Test the setup at this point

- Select the *rigidSpring_Constraint*.
- In the Channel Box or attribute editor enter a value into the **springRestLength** attribute that is **1/2** of its current value.

This will force the spring to try to assume this new length thus pulling the *lever* towards the *overhead* object.

8 Attach the boom to the boomWheel

You should have the *boomWheel* rotating correctly. You want the *boom* and its launcher to follow this rotation. To do this you will apply another orient constraint between the *boomWheel* and the *boom*.

- Select the *boomWheel* then the *boom* object.
- Select **Constrain** → **Orient**

  The *boom* is now orient constrained to the *boomWheel.*

  The *boom* also contains, as children, the geometry that will hold the projectile. We have named it the *launcherPad*.

9 Make the launcherPad into a passive rigid body group

- Select the *launcherPad* group which contains the launcher geometry (stop, pad).
- Make this group a passive rigid body group by selecting **Bodies** → **Create Passive Rigid Body**

Note: You can make a hierarchy of objects into a single rigid body group by selecting the group and performing the create rigid body command. As long as there are no existing rigid bodies in this hierarchy.

---

10 Create a bomb

- Create a piece of geometry and place it on the *launcherPad* with a tiny little bit of clearance between the bomb and launch pad.
- Turn this object into an Active Rigid Body by applying a gravity field to it.

11 Tune the throw

To tune the throw you will change the spring constraint attributes:

SpringStiffness to **200;**

SpringDamping to **5;**

Spring RestLength to **1**

For the springRestLength you have already tried a value that is one half its initial value. Other values will change the action but it is generally a good idea to keep this value constant and use the other values of **springStiffness** and **damping** to control the catapult strength and recoil.

Damping and friction settings on the *launcherPad* and the *bomb* will also play a part in the simulation. Try **Friction** of **3** and **damping** of **1** on both the *bomb* and *launcherPad* to get started.

Friction controls how much resistance occurs between surfaces. Damping can be thought of as air density.

Exercises:

Create more realism for the catapult

- Keyframe rigid body dynamic attributes to settle the recoil of the armature.

  Hint: (rigidSpringConstraint1.damping)

- Stop armature from going through the overhead geometry.
- Combine scenes spring.mb and catapult.ma to form a renderable spring object for the catapult.

SUMMARY

Rigid body constraints in Maya are an important part of working with dynamically animated geometry. Understanding the various advantages of using a specific constraint for a specific application can greatly affect how you approach a shot.

Realizing different methods for building and controlling hierarchy of objects through parenting and constraints is another critical talent.

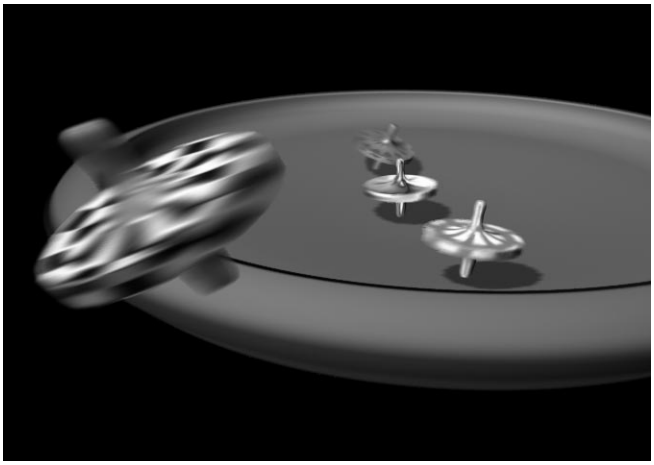
Chapter 3

3 Rigid Body Optimizing

This lesson covers the fundamental tools and techniques required to optimize scenes with Rigid Body components.

In this lesson, you will learn the following:

- Using stand-in geometry
- Rigid Body and solver optimization
- Collision and interpenetration troubleshooting
- Baking simulations



WHAT IS AN OPTIMIZED SCENE?

Question: When do you need to optimize a scene?

Answer: When it runs excessively slow.

Answer: When it fails due to interpenetration errors.

Answer: When it behaves erratically and unpredictably.

You generally expect performance problems when running simulations in dense environments and expect lighter scenes to present less of a challenge to the interaction and playback.

The first objective is to ensure that a scene progresses through the simulation only stopping or slowing down during intensive calculation.

The second objective is to achieve play back as close to *real-time* as possible without having to render or playblast the scene. This is a good goal but in many cases can not be attained due to hardware and software performance limitations.

To achieve these objectives you will look at some typical problem areas and the necessary steps to ensure that the solver is not driven into an unsolvable situation.

Unpredictable or wild results

This is typically caused by values being fed into the solver that are wildly changing or exceeding the proper expected range. The *proper expected range* are values that make sense for the solver on a given attribute or dynamic state. If the solver encounters a value that is much larger or smaller than it was expecting it may tell an object to travel much faster or farther than the other objects around it are prepared for.

The solver must make assumptions to increase its performance. When you intentionally or un-intentionally stress the simulation, you may exploit an assumption that the solver is making. This will result in errors. Remember the simulation is only an approximation.

The goal is to set the approximation to an appropriate trade off between accuracy and interaction.

Solver Grinding or Failing

When a scene is grinding or no longer making forward progress, it is time to take a look at what is going on more closely. The script editor is the first place to look for errors and information that may be produced by the solver. Even seemingly unimportant warnings can provide a clue to why a solver is failing farther on. Your first step should be to investigate all warnings and errors.

Learn to associate warnings and errors with symptoms and conditions that lead to solver problems.

Slow playback

In order to speed up playback the load on the solver must be reduced. The solver attempts to determine the solution as to where every rigid body vertice is at any given frame.

The first step most likely usually reducing the amount of vertices that the solver must keep track of. Using stand-ins will accomplish this as well as tuning rigid body tessellation to more coarse values.

Reducing the overhead of other scene components and display functions goes a long way to improving playback. Displaying in wireframe, hiding other unused objects etc...

Caching the animation is necessary for optimizing rigid bodies that have an unavoidable geometry density. Caching can use a lot of memory in scenes with dense geometry. Caching will not necessarily circumvent a difficult solution but can help troubleshoot and avoid waiting for solving of objects that have already been optimized.

Interpenetration Errors

An interpenetration error occurs when the solver can no longer guarantee the accuracy of the simulation due to geometry that has passed through another dynamic object. Typically the solver will stop or slow down the simulation if this error occurs.

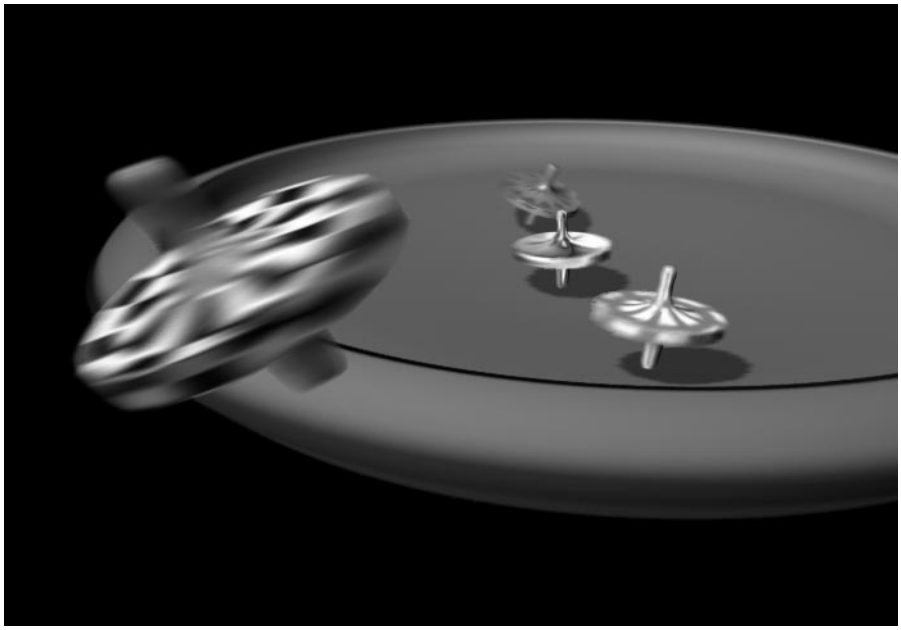
Why doesn't the solver just continue and ignore this error?

This error is important. You want this type of feedback. You can use this information to your advantage to help adjust simulation properties so that the solver does not enter into a corrupt situation.

Imagine if a sharp piece of geometry spiked into another object creating an interpenetration. The only way for the object to continue on would be to reverse its direction and back itself out then continuing in a less penetrating fashion. This would be a very laborious and computationally intensive moment. One that you want to avoid completely if possible.

Avoid Interpenetration errors by:

- Anticipating "at risk" objects.
- Using stand-ins
- Adjusting Tessellations
- Adjusting rigidSolver Attributes
    - Step Size
    - Collision Tolerance
- Damping to avoid wild velocities
- Surface normal orientations



Example: Battling Tops

This exercise steps you through some common scene problems that lead to optimization and troubleshooting.

1 Load the scene file

- Open the file *battleTop_start.mb.*

2 Playback the scene

The scene is set up to a point where it is in need of some optimization. It also has some “errors” creating problems. Ask yourself the following questions:

- What do you notice about performance?
- What do you notice about the base?
- What do you notice about the tops?

3 Create normals display on/off shelf buttons

- Select **Display** → **Nurbs Components** → **Custom** - ❒, and set the following:

  Normals to **On**

- Press **Save** to save the options.
- Press and hold **Cntrl+Alt+Shift** simultaneously then select **Display** → **Nurbs Components** → **Custom.**

  The menu item is now loaded on to the current shelf as a shelf button.

- Repeat the above process to create the off button with the **Custom**-❒ settings to **Normals** toggled **off**.

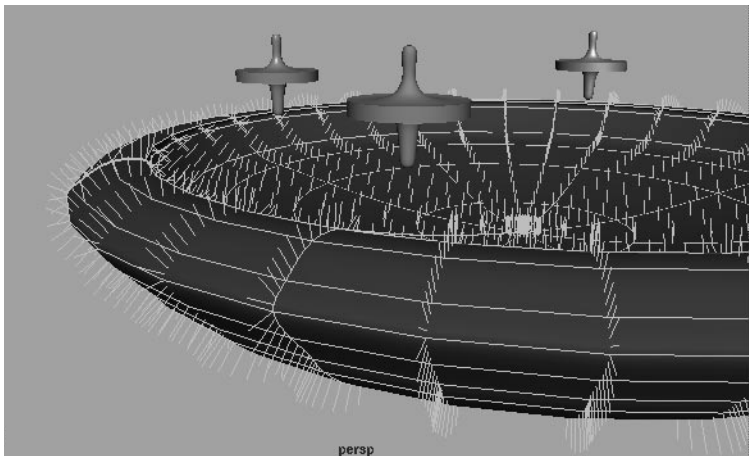
4 Select and display the object normals

- Select the base and the tops.
- Use your normals display shelf buttons to display the normals for the base and tops

  The base's normals are reversed. This is a common occurrence for objects that were created using surface modeling tools like revolve and sweep.

5 Reverse the base surface

- Select the base object.
- Select **Edit Surfaces** → **Reverse Surface Direction** - ❒, and press **Reset**.
- Press **Reverse.**



Normals Displayed with Correct Orientation

6 Reverse the surfaces of the Tops

Although this will improve performance and may alleviate the interpenetration problems you were experiencing, the scene is still running very slow.

- Repeat the previous steps for the tops surfaces.

7 Rebuild the Tops geometry

Another reason for hampered playback is geometry that is needlessly overbuilt, therefore you will rebuild the tops geometry.

- Select one of the *tops.*

- Select **Edit Surfaces** → **Rebuild Surfaces**- ❐, and set the following:

  Rebuild Type to **Reduce**

  Parameter Range to **0 to 1**

  Direction to **U and V**

  Use Tolerance to **Local**

  Positional Tolerance to **1**

  Output Geometry to **Nurbs**
- Repeat this rebuilding process for all the top.

Tessellation Factor for NURBs rigid bodies

When working with Nurbs geometry it is important to note that the Tessellation Factor attribute under Performance Attributes will control the polygonal approximation of the Nurbs surface. This attribute has no effect on Polygonal rigid body objects.

By default the tessellation factor is 200.

By rebuilding the Tops objects you have reduced the geometric load on the solver. Another way is to adjust this Tessellation Factor attribute. The main drawback being that you do not have control over where the tessellation is most needed. In this example we would want more tessellation at the tips to avoid a sharp interpenetrating point. And also at the sides to avoid interpenetration when tops collide.

Another strategy might be to convert your rigid bodies to polygons during the rebuilding process. Thus creating a stand in object for collisions and simulation.

Stand ins

One of the best ways to clean up the performance of a rigid body is to use a stand in object. This is very useful when you have dense or irregular objects that you want to perform rigid body dynamics to.

The stand in can also be a lower resolution object that will provide faster evaluation.

1 Substitute the base object with the lowRezBase object

One object in the scene is needlessly complex. The base object has geometry that is not involved in collisions.

- Select *base* object.
- In the Outliner, delete the associated *rigidBody*.
- Select *lowRezBase.*
- Select **Bodies** → **Create Passive Rigid Body.**
- In the Outliner, **Ctrl-select** the *lowRezBase* and the *tops* objects.
- Select **Solvers** → **Set Rigid Body Collision.**

The Solvers menu contains two commands:

Set Rigid Body Collision

Set Rigid Body Interpenetration

Set Rigid Body Collision controls which objects will collide for the current solver

Set Rigid Body Interpenetration controls which objects will not collide with one another for the currently selected solver.

These two menu items are actually executing the mel command `rigidSolver` with the following flags:

```
rigidSolver -e -collision object1 object2 (object...)
rigidSolver
```

```
rigidSolver -e -interpenetration object1 object2
(object...) rigidSolver
```

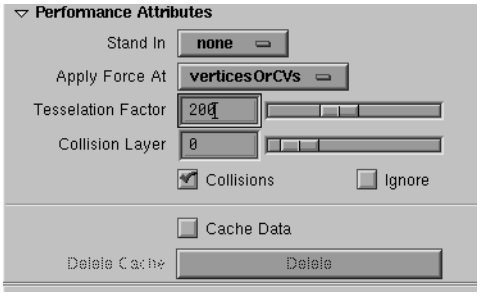
There are many options to the rigidSolver command. Look in the Documentation for the rigidSolver command under scene commands section found under the MEL Index.

2 Playback the scene

3 Set Tessellation for the tops rigid bodies

The rigid body dynamics are determined from an approximation of the rigid body objects shape. This approximation is controlled by the tessellation of the rigid body. Unless a stand-in object is specified.

- Select *top1* rigid body.
- Open the Attribute Editor and locate the Performance Attributes section.



Performance Attributes Section, Rigid Body

Stand In - Selects Stand-in Object Sphere or Cube

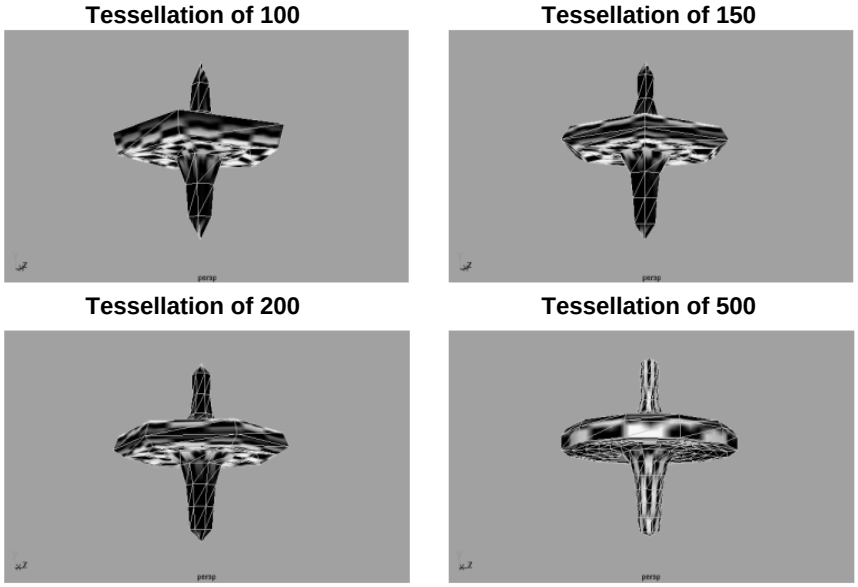
Apply Force At - Dynamic Forces applied at Center of Mass, Bounding Box or VerticesOrCVs.

Tessellation Factor - Adjust Geometric Approximation of rigid body object.

Collision Layer - Select Collision Layer participation.

- Experiment with extreme values on the **Tessellation Factor** attribute with **Stand In** set to **none**.

Below are the equivalent tessellations for the top1 object set to 100,150, 200, and 500.



Tessellation values from 100 to 500

Note that at low tessellation values the point of the top is much sharper. This will affect how the top will spin and also increase the odds of an interpenetration. Also at low tessellation values the interaction of the edge of the top with other tops will be less predictable.

Example: Windy City

In this scene you will work with multiple solvers and Bake Simulation to slim down a very busy alley way.

One of the setups in this scene is a trashcan that is falling down the stairway. This trashcan is to be filled with trash. This trash is to fall out and litter the alleyway.

Your first inclination should be, that seems like a problematic situation. Your second inclination should be one of problem solving before the problems present themselves.

One problem will be choreography. Getting all the pieces to behave in a realistic manner.

Another problem will be getting these interdependent objects reacting to each other without stalling the solution due to interpenetration.

1 Open scene file

This scene consists of an alley way with typical urban obstructions and a average dose of garbage.

- Open the file *alley.mb.*

2 Create two additional solvers

- Select **Solvers** → **Create Rigid Body Solver.**
- Select **Solvers** → **Current Rigid Solver** to confirm that a new solver was created.
- Repeat so that you now have 3 rigid body solvers:

  rigidSolver

  rigidSolver1

  rigidSolver2
- Select *rigidSolver1* as the current solver for the following steps.

  Subsequent rigid body actions taken will occur with the currently selected solver. So, creating rigid bodies and setting collisions etc... will run under control of the solver that is active when these actions are taken.

Animate the Trashcan

The trashcan presents some problems that we can alleviate by using a standin object. We will use a cylinder as a stand in object. Your initial goal here is to have trash inside the trashcan.

1 Create Passive Rigid Bodies out of the lowSteps, stairway and floor objects

- Verify that *rigidSolver1* is the current rigidSolver

  Select **Solvers** → **Current Rigid Solver** → **rigidSolver1.**
- Select the *lowStep_group*, *stairwayGroup* and *floor* objects
- Select **Bodies** → **Create Passive Rigid Body.**

  These objects are now rigid bodies under the *rigidSolver1* solver. By breaking up the duty of individual solvers you can optimize a complex scene further.

2 Use a standin object for the trashcan

- Un-hide the *trashcanStandIn* object.
- In the Outliner, drag the *trashcan* onto the *trashcanStandIn.*

  This parents the *trashcan* to the *trashcanStandIn.*

3 Turn the trashcanStandIn into an Active Rigid Body

- Select the *trashcanStandIn* objects shape node.
- Select **Bodies** → **Create Active Rigid Body.**

4 Connect this rigid body to the Air fields and gravityField1

- In the Outliner, select the *trashcan* rigidbody and **ctrl-select** the *airField1*.
- Select **Connect/Add** → **Connect to Field.**
- Repeat this step to connect the *trashcan* to *airField2* and *gravityField1*.

5 Adjust trashcan rigid body attributes to satisfy the simulation

We would like the *trashcan* to fall down the steps but also follow the curve so that the trashcan ends up at the bottom of the stairs and continues down the alley. Use the following settings as a guide. All other un-noted attributes are assumed to be default.

Rigid Bodies:

Mass to **10**

Bounciness to **0.4**

Damping to **0.1**

Static Friction to **0.2**

Dynamic Friction to **0.2**

Collision Layer to **0**

Stand In to **none**

Apply Force At to **bounding Box**

AirField1:

Magnitude to **7**

Attenuation to **0**

Use Max Distance to **off**

Speed to **0.5**

Enable Spread to **Off**

AirField2:

Magnitude to **5**

Attenuation to **0**

Use Max Distance to **off**

Speed to **1**

Enable Spread to **off**

Experiment with these settings. Some attributes can have a profound impact on the *trashcan's* movement.

Bake Simulation

Once you have a successful simulation of the trashcan, you can simplify things by baking the simulation. Baking will convert the dynamics induced animation to animation curves. These curves can then be tweaked and manipulated like any other animation curves in Maya.

The baking process will also allow for other benefits on performance and functionality. Once the trashcan object is baked via the trashcanStandIn we can then make the trashcan a passive rigid body which will allow for another layer of rigid body dynamics inside the trashcan.

1 Bake the trashcan falling down the stairway

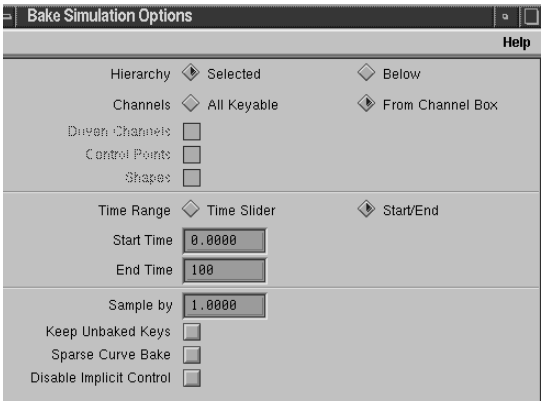
- Select the *trashcanStandIn* object.
- Select **Edit** → **Keys** → **Bake Simulation** - ❐, and set the preferences as follows:

Hierarchy to **Selected -** You only want to bake the selected object not the entire hierarchy.

Channels to **From Channel Box** - You will highlight the appropriate channels in the Channel Box.

Time Range to **Start/End** - You will bake only **100** frames first to get an idea of how it is working.

Sample by to **1.0** - You will sample by **1** frame increments.



Bake Simulation Window with correct preferences

- In the channel box select the **Translate** and **Rotate** channels.

Tip: **LMB** drag down the channel box across these attributes so that they are highlighted in Black.

- Press **Bake.**
- If all is well, then go back and bake off the entire simulation.

2 Ignore the trashcanStandIn rigid body

You will set the *trashcanStandIn* rigid body to **Ignore** and not obey collisions. You could delete this node but you never know when you might want to go back and re-bake a different simulation.

- Select the *trashcanStandIn* rigid body.
- In the Channel Box set the **Ignore** attribute to **on**
- In the Channel Box set the **Collisions** attribute to **off**.

3 Playback

Confirm that the bake is accurate. Problems with bake can arise when either you are sampling by too course of a sample rate and / or you are trying to bake too many channels from control points or shape node animation. Treat these other nodes and channels as separate passes.

Putting trash in the can

1 Make the trashcan a Passive Rigid Body

- Select the *trashcan* transform node.
- Select **Bodies** → **Create Passive Rigid Body.**

2 Add Active Rigid Body contents to the trashcan

In the *alley.mb* scene file is a group named *garbage*. This group contains various pieces of refuse. You will use them to test with. They have standin geometry already parented to them.

- For example, unhide the *paper* group and select an individual paper object. Unhide it, then position it inside of the *trashcan*.
- Make the stand-in paper object you selected an active rigid body by selecting **Bodies** → **Create Active Rigid Body.**
- Connect this rigid body to *gravity2* and *AirField2*.

3 Playback and tune for interpenetration

Damping, bounciness and **friction** will play a big part in preventing the active rigid body trash objects and trashcan from reaching accelerations that can cause interpenetrations. The solver attributes of **Step Size** and **Collision Tolerance** may also need to be adjusted.

Trash Object Rigid Body Settings:

Mass to **1**

Bounciness to **0.3**

Damping to **0.5**

Static Friction to **0.2**

Dynamic Friction to **0.2**

Collision Layer to **(separate integer values for each object)**

Stand In to **Sphere**

Trashcan object Passive Rigid Body Settings:

Mass to **10**

Bounciness to **0.1**

Damping to **0.5**

Static Friction to **0.01**

Dynamic Friction to **0.01**

Collision Layer to **-1**

Stand In to **none**

Collision Layers

Another method to avoid overly complicated collisions is to move the trash objects onto their own collision layer. The objects will not collide with each other, and this may be forgivable at this stage of testing. By setting the trashcan and the stairs and the floor to collision layer **-1**, they will still collide with the trash objects. The setting of **-1** means collide with all collision layers. Objects on separate solvers cannot collide regardless of their collision layer setting.

rigidSolver settings:

Step Size to **0.010**

Collision Tolerance to **0.1**

Scale Velocity to **1**

Start Time to **1**

Rigid Solver Method to **Runge Kutta**

The solver settings are very important to solution success. It is a good place to look at early in the process of getting a good solution. If your settings of **Damping** and **Friction** do not alleviate gross interpenetration problems then the solver settings of **step size** and **collision tolerance** should be adjusted.

The solver method will also greatly influence performance. The **Runge Kutta** method is Maya's default method and the best balance between accuracy and speed for most applications. Experiment with these methods so that you are comfortable with their respective strengths and weaknesses.

Moving objects to another rigidSolver

Using multiple rigid solvers gives you the ability to adjust solver settings and methods to suit individual rigid body needs.

To select the current solver:

- Select **Solvers** → **Current Rigid Solver** → **RigidSolver.**

To create a new solver:

- Select **Solvers** → **Create Rigid Body Solver.**

1 Duplicate floor object

- Duplicate the rigid body *floor* object and rename the new rigid body object *newFloor***.**

  If you want to create a new separate simulation of trash blowing down the alley it may be more efficient to create a new solver and run the objects on the new solver. Remember that objects cannot collide or be influenced by objects on separate solvers. You can, however, duplicate objects and put them on another solver or create them under a new solver.

2 Create a new rigid body solver

- Select **Solvers** → **Create Rigid Body Solve.**

  This solver is now the active solver. Any active or passive rigid bodies created will be created under this solver unless you select another solver as the current solver.

3 Move to rigidSolver1

You will move the *newFloor* object to *rigidSolver1*

- First select *newFloor's* rigidBody node then in the command field or the script editor enter:

  ```
  rigidBody -edit -solver rigidSolver1;
  ```

  You can select multiple rigidBodies then execute this command to move all over at once.

Note: See the Appendix section on creating the rigidBody shelf buttons to make this functionality always available without the typing.

This new solver and floor are now ready to work with newly created rigid bodies. This new solver can have separate settings while maintaining the settings already in place for the existing trashcan simulation.

Conclusion

Tuning and TroubleShooting rigid bodies can be broken down into a straightforward methodology.

Know your Tessellations - Surfaces or Polygons, what is Solver seeing in terms of geometry.

Which way are the Normals facing? - This is the number one reason for problems.

Will StandIns help the Solver work with this geometry?

Bake Simulation to add control to complex scenes.

Collision Layers are a good method of separating out objects that you do not want the solver to calculate collision between.

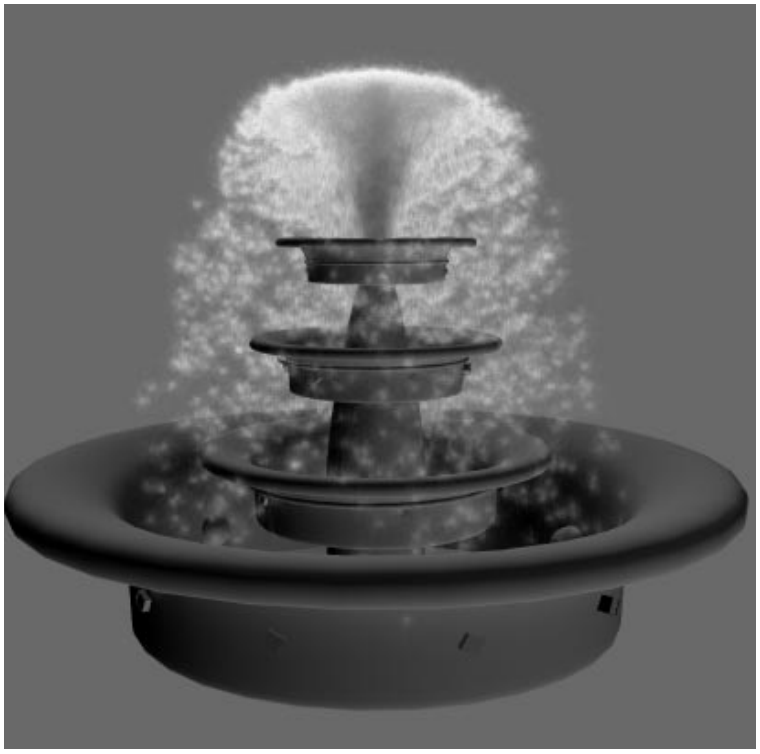
Using **multiple Solvers** can help localize control over a specific simulation or collision / interaction.

Chapter 4

4 Introduction to Particles

This lesson focuses on the basic concepts required to understand and work efficiently with particles in Maya. This lesson should be considered as an essential framework to build upon for more advanced concepts which are discussed later. The following areas will be addressed in this lesson:

- The particle shape node
- Basic particle attributes
- Emitters



The structure of particles in Maya

Particles in Maya differ from geometry in the following ways:

- Particles are points in space. They require special handling at render time because they do not contain surface information.
- Particles can be rendered using Hardware or Software rendering methods. The particle Render Type attribute controls which of these two methods are used.
- Individual particles belong to a common collection referred to as the *particleShape* object. Just as CVs of a geometric object belong to their shape node, individual particles can be thought of as components to the particleShape node.
- Particle attributes are commonly categorized into two types: Per Particle(array) and Per Object.

Applications of particles

Particles are commonly used to simulate complex natural phenomena. Common examples include smoke, rain, sparks, gases, dust, snow, fire, and other motion that consists of complex or random motion of many individual components.

Particles can be keyframed or controlled dynamically.

Creating Particle Clouds

There are several methods for creating particle objects in Maya. In this first example, the focus will be on using the Particle Tool.

The particle tool is a quick and easy way of creating individual particles, particle grids, and random collections of particles. This can be useful for just getting some particles to start working with. This also provides you with the ability to interactively place particles exactly where you want them.

1 Create a cloud of particles

- Select **Particles** → **Particle Tool** → **Options -** ❐, and set the following:

  Particle Name to **particleCloud**;

  Number of Particles to **50;**

  Maximum Radius to **10**

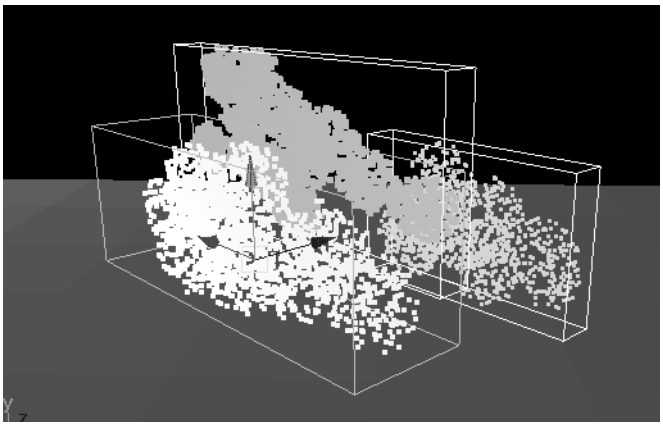
- Click in several locations in the viewport to place particles in the scene.

  Each click-release produces 50 particles randomly within the volume of an invisible sphere of radius 10 units.

Tip: If you drag while the mouse button is depressed, you can interactively change the placement of where the particles will be dropped. The particles will be placed at the location where the mouse button is released.

- Press **Enter** when finished placing particles.
- Open the Outliner to see the newly created particle object called *particleCloud*.

  You will need to show shapes in the Outliner to see the *particleCloudShape* node.
- Enable Outliner option; **Display** → **Shapes**.



Creating 3 clouds of particles with the particle tool

2 Create several cloud shapes

You will animate a few layers of clouds.

- Repeat the above steps to create some additional particle objects in the shape of clouds.

3 Animate the particleCloud transform nodes with keyframing

Traditionally particles have been animated exclusively through fields and dynamic expressions. In Maya you also have the option to animate the particle objects transform like any non-dynamic object. Translate, Scale and Rotation can be keyframed to provide many common effects.

- Select the transform node of each particle object.
- Translate, Rotate, and Scale as needed to position and size each cloud.
- Set keyframes on transform attributes as needed to make a short animating cloud example.

Tip: To add the Particle Tool to a shelf as a shelf button, simultaneously press and hold **Shift**+**Control**+**Alt,** then select **Particles** → **Particle Tool.**

Note: It is also possible to create an *empty particle object* by setting the number of particles to 0 in the particle tool options. Empty particle objects are commonly needed when working with particle emitters.

EMITTERS

An emitter is like a cannon that projects particles into space. Below is a list of the different kinds of emitters available in Maya.

- Directional
- Omni Directional
- Curve
- Surface
- Texture
- Per Point

Open the file *emitterTypes.ma* to see an example of each of these.

Display in shaded mode is recommended to see the textured particles.

Create a simple fountain using a directional emitter

1 Load the scene file

- Open the file *fountainGeo.mb*.

  This file contains geometry of a simple fountain with no dynamics. You'll add a directional emitter and modify some of its attributes.

2 Create a directional emitter

A directional emitter allows you to specify exactly what direction in world space to emit the particles.

- Select **Particles** → **Create Emitter** -❐, and set the following:

  Emitter Type to **Directional**

- Press **Create.**

  This creates two new objects. *Emitter1* is the directional emitter that emits particles into the particle object *particle1*.

3 Position and name the emitter

- Select *emitter1* in the Outliner or persp window then position it slightly below the tip of the fountain's *spout*.
- Rename *emitter1* to *spray*.
- Rename *particle1* to *droplets*.

4 Playback the animation

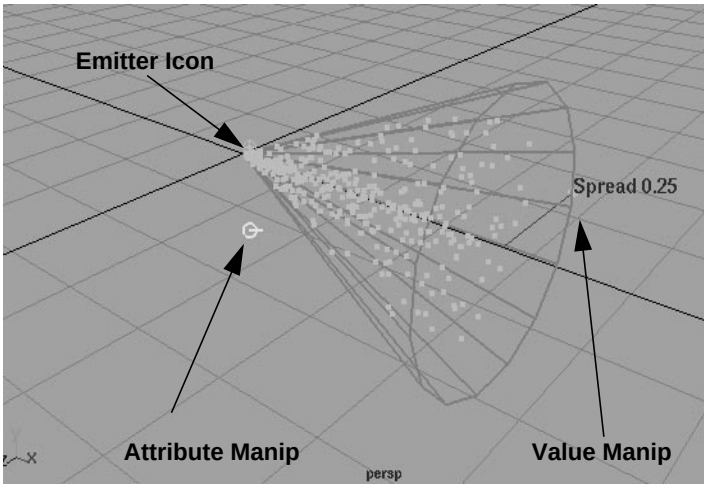
- Set the playback range to start at frame **1** and end at frame **500** then playback the animation.

  By default the particle emitter direction occurs along the X axis.

5 Modify the Emitter's attributes using manipulators

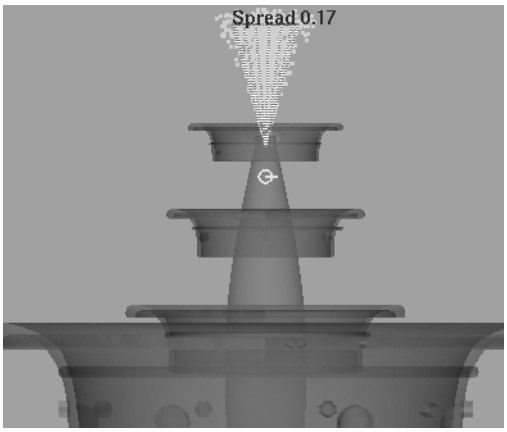
- With the emitter selected, press the **t** key to switch to the Show Manipulator Tool.

  The small circular icon below the emitter is a toggle switch that cycles the manipulator through different attributes on the emitter so that each attribute can be quickly edited graphically. This manipulator functions similar to manipulators on spotlights which you may already be familiar with. The *value* manip will change based on the Attribute manip selection. It can be click dragged to change the attribute value.



Using the show manipulator tool with an emitter

- Use the manipulators or Channel Box to adjust the **direction**, **speed, rate**, and **spread** of the emitter until the result resembles a fountain as shown below.



The spray directional emitter with its manipulator displayed

- The following values work well for the *spray* emitter:

  Direction to **0, 1, 0;**

  Rate to **1000;**

  Spread to **0.17**

Note: A **spread** value of 1 corresponds to a 180 degree emission cone. **Rate** is a measurement of the number of particles per time unit that are emitted. The default time unit is seconds. **Speed** determines how fast the particles leave the emitter.

Fields

There is a subtle but important difference between *creating* a field (i.e. **Create Turbulence**) and *adding* a field (i.e. **Add Turbulence**). When a field is *created*, it is not placed within any object hierarchy. Instead, it is placed at the origin. When a field is *added* to an object, one field gets parented under each object that was selected at the time the field was added.

6 Change the particle render type to MultiPoint

- **Click-Drag** select *droplets* in the perspective window.
- Open the Attribute Editor, and set the following:

  Render Type to **MultiPoint**

  Depth Sort to **On**

- Press the **Current Render Type** button and set the following:

  Color Accum to **On**

  Use Lighting to **On**

  These options determine the draw, shading, and lighting properties of this particle object. These settings make the particles appear somewhat more like water droplets in motion. The specific details of these options will be discussed more in the rendering chapter.

7 Add gravity and turbulence to the droplets

Adding fields will help define the particles' motion better.

- Select *droplets.*
- Select **Fields** → **Create Gravity.**
- With the particles still selected, select **Fields** → **Create Turbulence.**
- Playback the scene to test the gravity and turbulence fields.

8 Increase spray's speed attribute

Using a higher speed value causes the particles to leave the emitter faster which, in turn, allows them to travel higher before being overcome by gravity.

- Select *spray*.
- Set **Speed** to **10** in the Channel Box.

Add a secondary particle object

To give the impression of streaks in the water you'll now add a second particle object for the emitter to emit into. The attributes of this particle object can be controlled independently from the *droplets* particles.

1 Use the particle tool to create an empty particle object

An "empty particle object" is created when the **Number of Particles** setting in the Particle Tool's options is set to 0. This creates an empty storage place for the emitter to later emit into.

- Select **Particles** → **Particle Tool -** ❐, and set the following:

  Set **Particle Name** to *mist*.

  Set **Number of Particles** to **0.**
- Click anywhere in the perspective window.
- Press **Enter** on the keyboard.

  You may receive a warning that no particles were created. This is normal.
- Check the Outliner to make sure the *mist* particle object was created.

2 Establish emission for mist

Currently, there is no relationship between *mist* and *spray*. You'll now establish a connection between the two.

- Select *mist* in the Outliner.
- **Ctrl-click** to select *spray*.
- Select **Connect/Add** → **Connect to Emitter.**

  Rewind and playback. The *spray* emitter now emits into both *mist* and *droplets*.

3 Set display attributes for mist

Just as you defined descriptive attributes for *droplets*, you can do the same for *mist*. Use the Attribute Editor to set the following attribute values for *mist*:

Render Type to **Multi-Streak**

Depth Sort to **On**

Color Accum to **Off**

Use Lighting to **On**

Setting **Color Accum** off for *mist* creates contrast against the *droplets* particles whose **Color Accum** setting, you'll recall, was on.

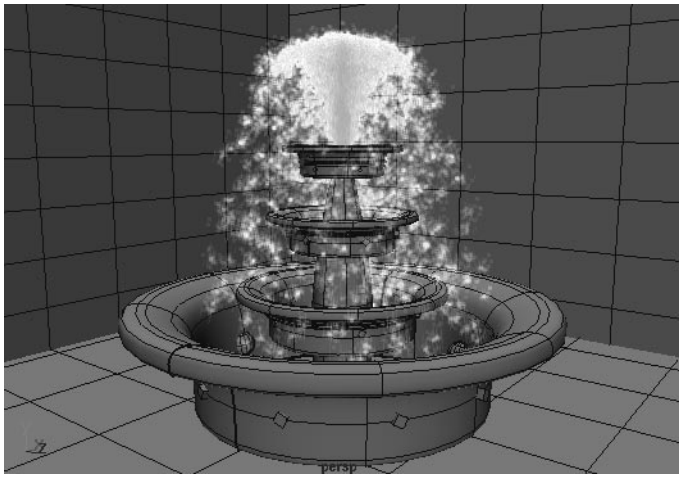
Tip: When **Color Accum** is on, overlapping particles within the same particle object get their RGB values added together. This creates a more "washed-out" or additive appearance.

---

4 Connect mist to the existing gravity

- Select *mist* in the Outliner.
- **Ctrl-select** *gravity1* located inside the *fountain* group.
- Select **Connect/Add → Connect to Field.**

When played back, the *mist* particles should fall with *droplets.*



The spray directional emitter emitting into particle objects droplets and mist

Understanding particle attributes

The next step is to add more specific controls to *droplets* and *mist*. This requires a clear understanding of some important concepts that will be discussed here briefly before continuing with the fountain.

- The most commonly used particle attributes exist on the *particleShape* node, The Transform node contains the traditional transform attributes (*translate*, *scale*, *rotate*, etc...)
- All particle objects use *position*, *velocity*, *acceleration* and *mass* attributes. Therefore, these attributes are part of the particle shape node by default and cannot be deleted.
- There are many other attributes (*lifespan*, *radius*, *color*, *incandescence*, etc.) which can be added to particles. This allows you to customize each particle shape to your specific needs and also keeps things more efficient.

- Some attributes are intended to be used for only specific particle render types. For example, *spriteNum* is only intended to be used with sprite particles.

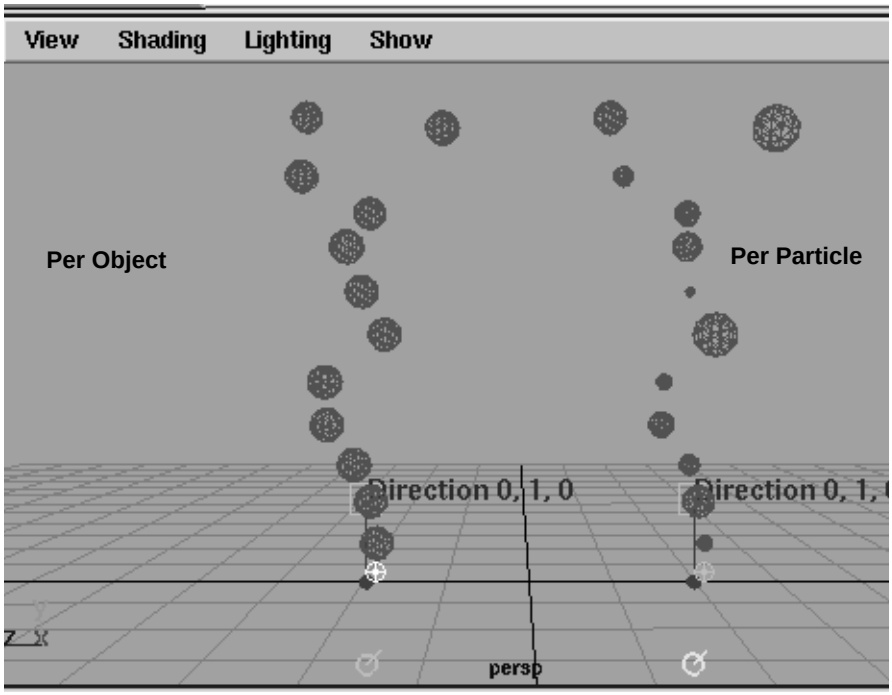
Per Particle vs. Per Object Attributes

It is important to understand the difference between per particle and per object attributes.

- **Per particle** attributes allow each particle to store its own value for a given attribute.
- **Per object attributes** assign one attribute value to the entire particle object.

  It has become a common convention to name per particle attributes with a PP at the end (radiusPP, rgbPP, etc). However, it is not an absolute requirement.

Tip: There is extensive information regarding various particle attributes in the *Maya Dynamics* and *Dependency Graph Node* sections of the on-line documentation.



Per Object vrs. Per Particle radius

In the picture above, the particles emitted from the left emitter were given a per object radius attribute (radiusPP). The particles emitted from the emitter on the right are in the same relative position as those emitted from the emitter on the left. However, each particle has its own radius value.

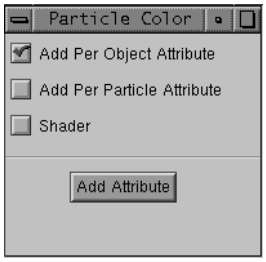
Color and Lifespan attributes

You will continue working with the fountain you have just created.

1 Add a per object lifespan and color attribute to the particles

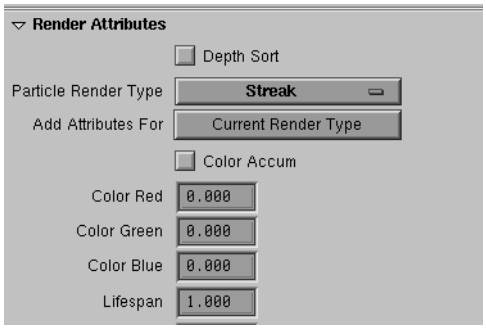
Adding a lifespan attribute gives you control over how long the particles stay in the scene before they disappear. For now, you'll assign the same lifespan value to all the particles to keep things simple.

- Select *droplets* and open the Attribute Editor.
- In the **Add Dynamic Attributes** section, press **Color** and select **Add Per Object Attribute**.
- Press **Add Attribute.**
- In the **Add Dynamic Attributes** section, press **Lifespan** and select **Add Per Object Attribute**.
- Press **Add Attribute.**



Adding a per object attribute

Fields for editing **RGB** and **Lifespan** are added in the Render Attributes section (and to the Channel Box) for this particle object as shown below.



Changing the per object RGB values in the Attribute Editor

- Set the following attributes for **Color** and **Lifespan**:

  Color Red to **0.5**;

  Color Green to **0.5**;

  Color Blue to **1.0;**

  Lifespan to **2.6**

2 Repeat the process for mist particles

- Create an Add Per Object Lifespan and Color attributes to *mist*. Use the same Lifespan value but modify the RGB values very slightly to add some variation.

3 Test the animation

- Rewind and playback the animation.
- If necessary, adjust the lifespan so the particles die near the time they are at the same Y coordinate as the bottom pool of the fountain.
- Select the particles press **5** then **7** on the keyboard.

  Now when you playback the animation, the particles are displayed with color and lighting.

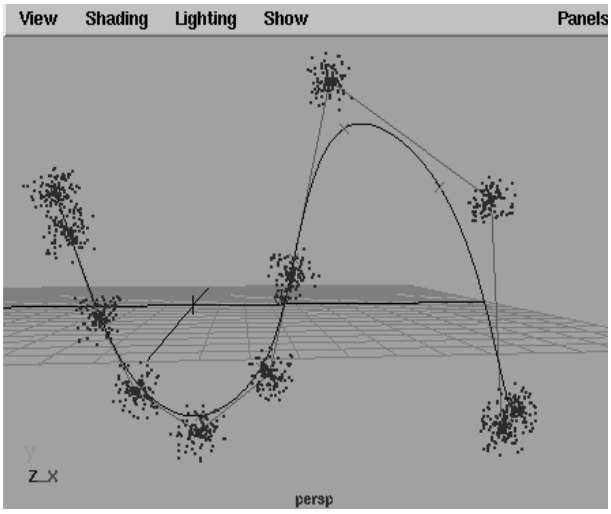
  For better results when rendering this fountain, increase the *spray's* **rate** to **2000** and also add a opacity attributes to the spray and droplets.

  You may notice that the particles do not collide with the fountain in this example. Particle collisions will be discussed in a later chapter.

Note: The units for lifespan is seconds so it is important to be aware of your frames per second settings and the different results that can occur if the same file is used on two different machines with different fps settings. To check the FPS setting open, **Options** → **General Preferences** → **Units.**

Omni-Directional Emitters

Omni-directional emitters can be added to NURBS, polygons and curves. An Omni-directional emitter causes particle emission to occur from the CVs (or vertices) of the object it is added to. The emission from that point emanates equally in all directions as opposed to only one specific direction which is the case with the previously discussed Directional emitter.



An Omni-Directional emitter emitting from a curve's CVs

Curve Emitters

Although it is possible to add a Directional or Omni-directional emitter to a curve, doing so will cause emission to occur only from the CVs of that curve, not from the portions of the curve between the CVs points and not from points on the curve.

A curve emitter is designed to allow omni-directional emission to occur along the entire curve instead of only from the CVs.

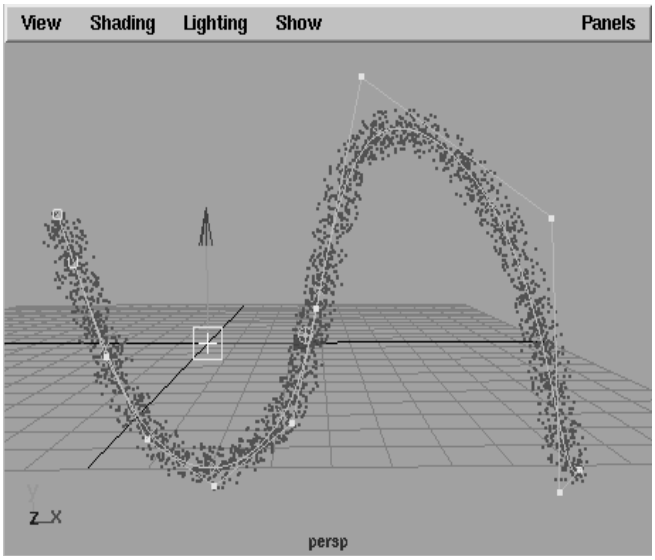
1 Create a simple curve emitter

- Create a NURBS circle or create your own edit point curve using the EP Curve tool.
- Select the curve and add a curve emitter by selecting **Particles** → **Add Emitter -** ❐, and set the following:

  Type to **Curve**.

2 Test the animation

- Playback the animation to view the curve emission.



The same curve with curve emission instead of omni-directional

Tip: It is possible to change the emitter type after the emitter has been created by editing the **emitterType** attribute in the Channel Box or Attribute Editor for the selected emitter.

Example: Whitecaps

Curve emission with per particle attributes

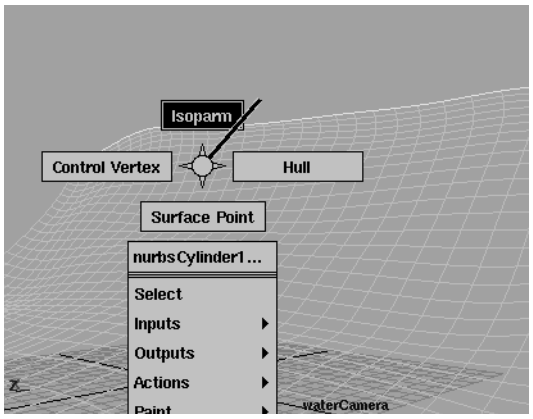
In some cases, it is useful to duplicate isoparms on surface geometry and use the resulting curve as an emitter. The steps below illustrate a quick example of this workflow and introduce you to using per particle attributes.

1 Open the scene file

- Select **File** → **Open.**
- Select *basicWave.mb*

2 Add a curve emitter to the edge of the wave

- Playback the animation to see the slight ripple motion that has been added to the object using a non-linear deformer.
- Rewind to frame **1**.
- Use the **RMB** menu to select the isoparm at the breaking edge of the ocean wave as shown below.



Selecting the edge isoparm of the ocean wave

- Select **Edit Curves** → **Duplicate Surface Curves**.

  A NURBS curve matching the shape of the selected isoparm is created.
- **Select** the NURBS curve.
- Add a curve emitter to it by selecting **Particles** → **Add Emitter**.
- Name the resulting particle object *foam* and the resulting emitter *foamEmitter***.**

Tip: You can also add curve emitters to a curve on surface (COS) object. You can create a COS using tools like Intersection or Project Curve or by making the surface *live* (**Modify** → **Make Live**) then drawing a curve with the EP or CV curve tools.

---

3 Adjust and add a lifespan attribute

- Select the *foamShape* particles and change the particle **Render Type** to **Cloud** in the Channel Box or Attribute Editor.
- Add a per object **Lifespan** attribute.

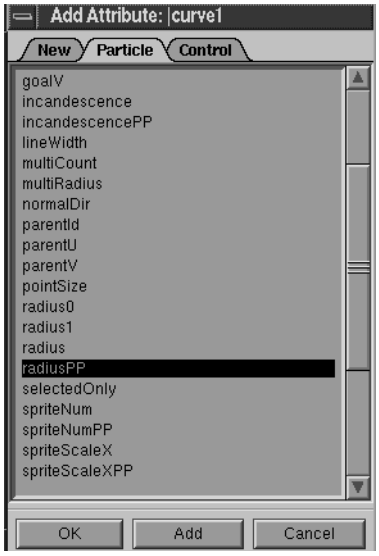
  A lifespan field is added. The default setting of **1.0** is fine for now.

4 Add a radiusPP attribute

Adding a radiusPP attribute will provide control over the radius of each cloud particle emitted.

- Open the Per Particle (Array) Attributes section of the Attribute editor.
- Press the **General** button in the Add Dynamic Attributes section of the Attribute Editor.
- Click on the **Particle** tab.
- Select **radiusPP** from the list of particle attributes.
- press **OK.**

A **radiusPP** attribute field is added to the **Per Particle (Array) Attributes** section of the Attribute Editor.



Adding a radiusPP from the Add Attributes window

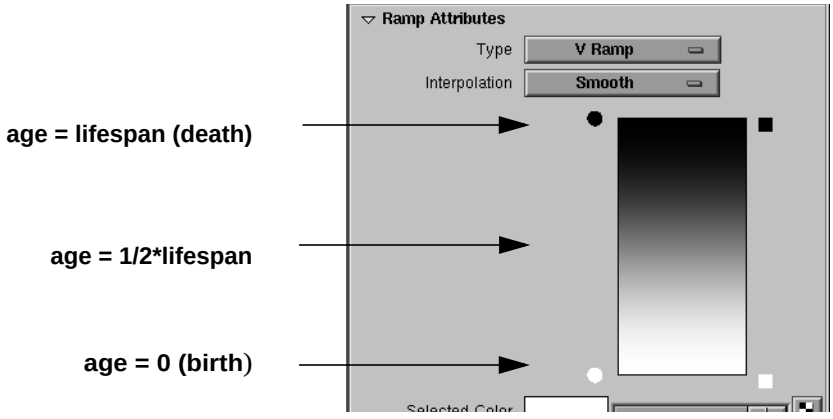
5 Add a ramp to control each particle's radius

- Click and hold **RMB** on the **radiusPP** field.
- Select **Create Ramp** from the pop-up menu.
- Click and hold **RMB** on the same **radiusPP** field and select **arrayMapper1.outValue1PP → Edit Ramp.**

  The ramp editor is displayed in the Attribute Editor.

6 Edit the ramp color

- Edit the ramp so there are only two color entries - white at the bottom of the ramp and black at the top as shown below:



A ramp added to radiusPP to control each particle's radius over age

The vertical "axis" of the ramp represents the particle's *normalized age*. The bottom of the ramp corresponds to the particle's birth, the top of the ramp corresponds to its death.

Normalized age

Normalized age is the relationship between a **particle's age** and its **lifespan** (age / lifespan). When a particle's age is equal to its lifespan, the particle dies.

Black corresponds to a radius value of 0, white to a value of 1. Therefore, with the current ramp configuration, the radius will be 1when the particle is born and 0 when the particle dies.

Tip: In order to assign values greater than 1 for float attributes such as **radiusPP** with a ramp, it is necessary to switch to **RGB** mode in the Color Chooser then enter the desired value in the **R** field. The numbers for **B** and **G** are ignored unless you are controlling a vector attribute.

- Set **Interpolation** to **Smooth** to provide a soft transition from black to white.

7 Test the animation

- Rewind and playback the animation.

  The radius decreases smoothly over the particle's age.
- Using these techniques, try to make the particles emit with a radius of **1.2**, stay at **1.2** until half way through their life, then linearly decrease to a radius of **0.1** when they die.
- You may want to decrease the emission rate while you adjust the effect then boost it back up when you've got the values to your satisfaction.
- Save the file as a mayaAscii file called *wave1.ma***.**

Tip: Adjusting **Noise** and **Noise Frequency** in the ramp is an effective way to achieve randomness. This works better with some attributes than others.

---

8 Adjust the Inherit Factor of the particles

Inherit factor controls how much of the emitting object's velocity is transferred to the particles during emission.

- Select the emitted particles and set **Inherit Factor** in the Channel Box to **1.0.**
- Rewind and playback the animation.

  When **inherit factor** is **0**, the particle velocity is not affected by the wave's deforming motion. A setting of **1** causes the particles to emit with the same velocity the moving curve has at the time of emission.

  If **inherit factor** was set to **0**, the particles would immediately fall down (due to gravity) even if the curve emitting them was moving up.

  Curve emission is also useful for simulating effects like shockwaves or energy pulses by adding the curve emitter to a curve and scaling the curve over time. This could be a stand alone effect or a greyscale rendering used as a displacement effect in compositing.

Surface Emitters

Surface emitters can be applied to NURBS and polygonal surfaces and cause emission directly from the surface rather than just from the CVs.

1 Load the file and switch to wireframe mode

- Load the file *glassEmit.ma.*
- Press **4** to ensure you are in wireframe display mode.

2 Add a surface emitter

- Use the Outliner to select *glassBase*.
- Choose **Particles** → **Add Emitter** - ❐, and set the following:

  Emitter Type to **Surface**;
- Press **Add**.

  This creates a surface emitter and parents it to *glassBase.* A particle object is also created.
- Name the emitter *baseEmitter* and the particles *baseParticles.*

3 Switch the render type to Sphere

- Select *baseParticles* and open the Attribute Editor.
- In the Render Attributes section of the Attribute Editor, set the following:

  Particle Render Type to **Sphere**.

  Press **Current Render Type** and set **Radius** to **0.02**

4 Test the animation

- Playback the animation

  The particles emit from the base towards the top of the glass but remain in the scene indefinitely.

5 Add and adjust turbulence for baseParticles

Turbulence will add some fluctuation to the movement of the particles to add realism.

- Select *baseParticles.*
- Select **Fields** → **Create Turbulence.**
- Select the *turbulence* field and position it near *glassBase*.
- Use the Channel Box to set **Attenuation** to **2** for the turbulence field.
- Increase **magnitude** to **8** and **frequency** to **2**

6 Add a per object lifespan attribute

- Use the techniques learned in the fountain example to **Add a Per Object Lifespan** attribute to *baseParticles.*

  Choose a lifespan value that causes the particles to die before they reach the top of the glass.

Attenuation

Attenuation controls an exponential relationship between the strength of the field and the distance between the affected object and that field. Imagine a curtain being blown by the air from a fan. In reality, as the distance between the fan and the curtain increases, the affect of the air from the fan on the curtain diminishes. It is this relationship between distance and field strength that attenuation controls. An attenuation of 0 causes a constant force regardless of the distance between the field and the affected object.

7 Add color to the baseParticles

- Use the Attribute Editor to add an **rgbPP** attribute to *baseParticles*.

Note: If an **rgbPP** attribute is added without a lifespan present, Maya will automatically add one.

- Add a *ramp* to the **rgbPP** attribute.

  Don't edit the colors or position of the color entries yet.
- Select the *drinkingGlass* object from the Outliner then template it using **Display** → **Object Components** → **Template**.
- Press **5** to switch to shaded mode.

8 Test the animation

- Rewind and then play the animation.

  As a particle's age approaches its lifespan, its color corresponds to a color higher along the vertical "axis" of the ramp editor.
- Edit the ramp so the color smoothly interpolates from white to a slightly light blue tint.

Normal Speed and Tangent Speed

Two attributes that are noteworthy when working with surface emission are **Tangent Speed** and **Normal Speed**.

These are closely related to the speed attribute that was previously discussed. **Normal speed** controls the particle's speed along the point that is normal to the point of emission. **Tangent speed** controls the particle's speed along a randomly selected point that is tangent to the surface of emission.

Tip: Setting both the **Normal Speed** and **Tangent Speed** to **0** is an easy way to get the emitted particles to stick directly on the surface the emission is occurring from.

---

1 Add water droplets to the glass surface and make them stick

- Select *drinkingGlass* and untemplate it.
- Add a surface emitter to *drinkingGlass***.**
- Rename the new emitter *glassEmitter*.
- Rename the corresponding particle object *glassParticles*.
- Set the following attributes for the *glassEmitter*

  Rate to **1;**

  Tangent Speed to **0;**

  Normal Speed to **0**

2 Test the animation

- Rewind and then play the animation.

The emitted particles “stick” to the glass since there is no normal or tangent velocity. However, there are too many particles stuck to the glass. The rate is as low as you can set it without using decimal values. If the rate is set to 0 no particles will emit.

3 Adjust the Max Count attribute

To limit the emission adjust the **Max Count** attribute.

- Open the Attribute Editor for the *glassParticles*.
- Set **Max Count** to **50** near the top of the Attribute Editor.

4 Test the animation

- Rewind and play the animation.

  The **max count** attribute limits the total number of particles the selected particle object is allowed to hold.

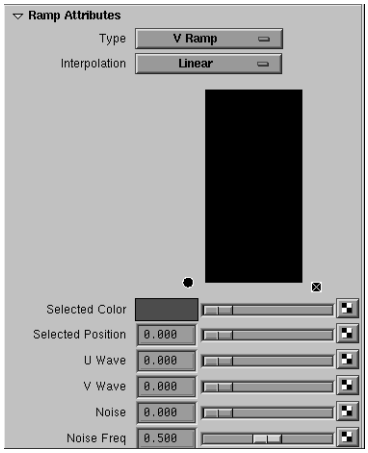
Note: When a particle object is created, **Max Count** is set to **-1** by default. This means Maya sets no limits on the number of particles it can hold.

---

5 Use a ramp to control acceleration

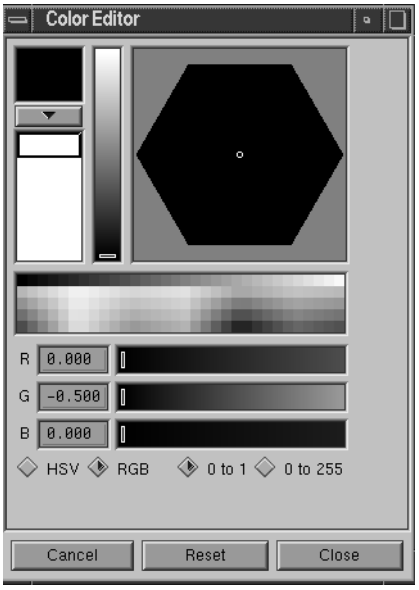
Currently, the *glassParticles* remain stationary during playback. It is easy to control their downward acceleration using a ramp. This is an alternative to attaching a gravity field.

- Open the Attribute Editor for *glassParticles***.**
- In the Per Particle (Array) Attribute section, **RMB** in the **rampAcceleration** field and select **Create Ramp** from the pop-up menu.
- Adjust the ramp so there is only one color entry (circular handle)



Acceleration ramp with a single color entry marker

- Move the color entry to the bottom of the ramp.
- Set the **RGB** values for the color entry to **0, -0.5, 0**. respectively



Setting a vector quantity (acceleration) using RGB

Note: When using the ramp editor to control **vector** quantities such as position, acceleration, and velocity, RGB corresponds to the particle's XYZ respectively.

6 Test the animation

- Rewind and play the animation.
- Experiment with different **RGB** values to see how the acceleration of the particles is affected.

TEXTURE EMISSION

With surface emitters, it is possible to control emission rate and location based on characteristics of a texture file or any 2D procedural texture. Texture emission works with textures only, not materials (blinn, phong, etc. or layered shaders).

Coloring particles using textures or procedurals

1 Load the file

- Load the file *paperBurn.mb.*

2 Add a surface emitter

- Select *emitPlane* then choose **Particles** → **Add Emitter** - ❐, and set the following:

  Emitter Type to **Surface;**

  Emitter Name to **textureEmitter.**

- Name the associated particle object *textureParticles.*

3 Set Tangent and Normal Speed

- Select *textureEmitter*.
- Set the **Tangent Speed** to **0.5** and **Normal Speed** to **1.0.**

  This prevents the particles from emitting directly normal to the surface providing for a more randomized emission appearance.

4 Add a rgbPP attribute to textureParticles

An **rgbPP** attribute is required for the inherit color option of texture emission to work. This is a common oversight when setting up texture emission.

- In the Attribute Editor for *textureParticles* add a **rgbPP** attribute.

5 Set the rate of textureEmitter

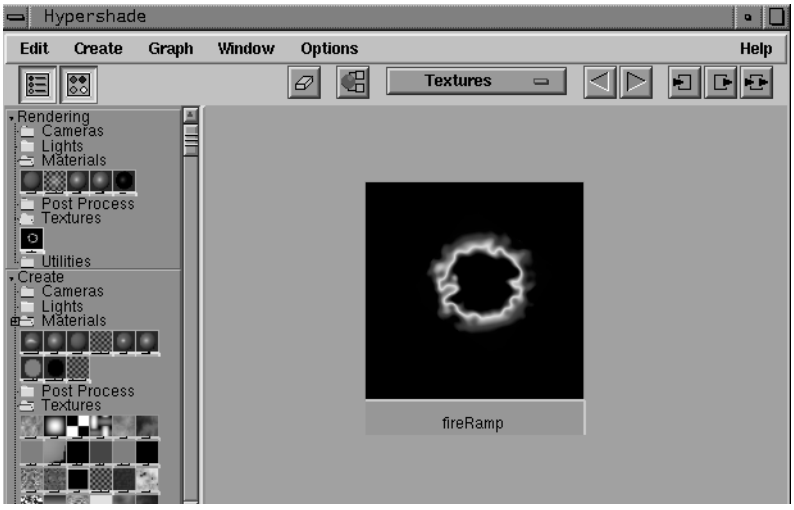
- Select *textureEmitter*.
- Set the **Rate** to **200** particles per second in the Channel Box.

6 Specify a texture for coloring the particles

Now you will connect a pre-made ramp texture to the emitter which will determine the color of the particles emitted.

- Open the Attribute Editor for *textureEmitter*.
- Select **Windows** → **Hypershade** and position it next to the Attribute Editor.
- In the Hypershade, Choose **Textures** from the menu bar currently labeled **Materials.**

  This displays all of the textures currently in the scene file. An animated fire ramp texture has already been prepared for this example and should be visible in the Hypershade window.



Displaying the fireRamp texture in the Hypershade window

- Locate the Texture Emission Attributes section of the Attribute Editor.

- With **MMB**, drag and drop the *fireRamp* texture icon from the Hypershade window onto the **Particle Color** Slider in the Attribute Editor for *textureEmitter*.
- Close the HyperShade Window.
- Turn **Inherit Color** to **On** in the Attribute Editor or Channel Box for the emitter.
- Press **6** to switch to hardware texture mode.

7 Test the animation

- Rewind and play the animation.

  Notice the emitted particles have inherited the RGB values from the texture at the location where they were emitted. Notice that black particles are emitting from the outer edge of the texture where the color is not present. You will be fixing this shortly.

Using a animated ramp texture to color the emitted particles

Note: To use a 3-D texture for texture emission, you'll first need to convert it to a 2D UV mapped image using **Convert Solid Texture**.

8 Add a per object lifespan attribute

- Add a per object Lifespan attribute.
- Set the **Lifespan** to **0.3.**

9 Add opacity so the particles fade out

Now you'll add opacity and a ramp to control it based on the age of the particles.

- Create a **Per Particle Attribute** for **Opacity**.
- Add a ramp to **opacityPP.**
- Edit the ramp so it has the following properties

  Color entry 1: **Selected Position** to **0**; **RGB** to **0.9, 0, 0**

  Color entry 2: **Selected Position** to **0.5**; **RGB** to **0.4, 0, 0**

  Color entry 3: **Selected Position** to **1.0**; **RGB** to **0, 0, 0**

Scaling Emission Rate with a texture

Similar to how you mapped a color image to control the color of emitted particles, you can also use the values of a greyscale image as multipliers on the emission rate of the emitter. This provides control over which portions of the surface will emit more than others. In this example, you want to shut off emission from the black areas of the ramp texture and leave emission on for the areas where there is color in the burning ring.

1 Connect a ramp to the rate

Since the areas of emission that you are interested in controlling correspond exactly to the color ramp that is being animated, the same ramp can be mapped to Texture Rate.

- Use the same method previously used to drag and drop the *fireRamp* from the HyperShade window. This time, drag it onto **Texture Rate** slider of the emitter.
- Turn **Enable Texture Rate** to **on**

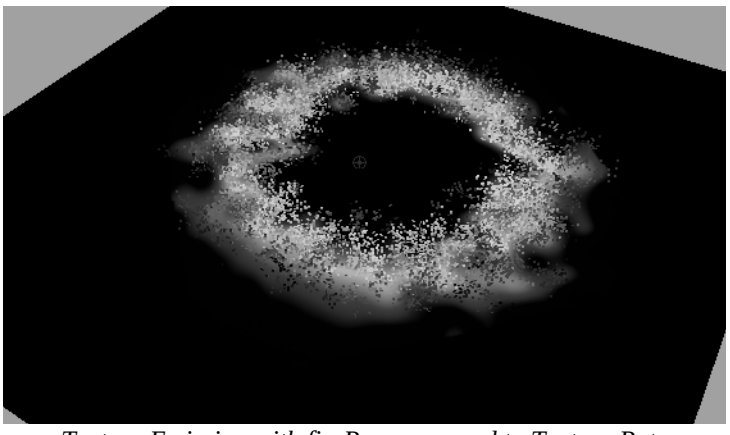
2 Test the results

- Rewind and playback

  Although fireRamp is a color image, when mapped to Texture Rate, Maya extracts the luminance values of the color ramp and uses those values as multipliers against the emission rate at the corresponding location of the surface.

  Areas of the color map with 0 luminance (black) use a 0 emission rate, areas of the color map with luminance of 1 use 100% of the emitter's rate.

  Once rendered, this fire ring could be rendered with a matte channel and combined with other textured surfaces using compositing software. These techniques will be discussed later in the course.



Texture Emission with fireRamp mapped to Texture Rate

Per-Point Emission

A common need when working with particles is to have some points on a curve or a surface emit at a different rate than other points. This can be

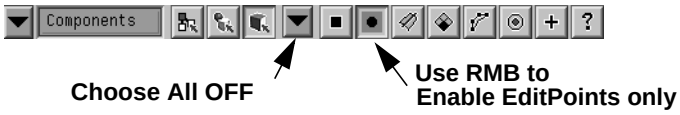
achieved to some extent using texture emission controls. Another option that offers some additional control is **per point emission**.

1 Open the file

- Select **File** → **Open.**
- Load *basicWave.ma*

2 Add omni emission to the edge edit points

- Use the technique learned earlier to duplicate the edge isoparm of *oceanWave*.
- Name the duplicated curve *waveCurve*.
- Select *oceanWave* then choose **Display** → **Object Components** → **Template.**
- Select the *waveCurve* then press **F8** to switch to Component mode.
- Set the pick mask so only **Edit Points** are available for selection.



Setting the pick mask for edit points only

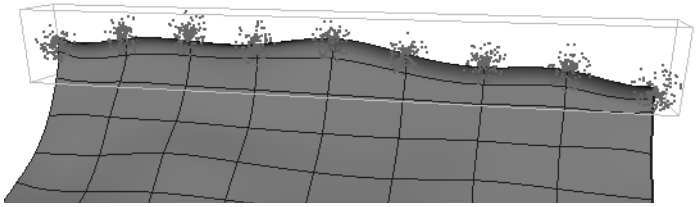
- **Drag-Select** around the entire *waveCurve* in the perspective window to select all of its edit points.
- Select **Particles** → **Add Emitter**
- Rename the emitter *perPointEmitter*.

  Notice that the **useRatePP** attribute of *perPointEmitter* in the Channel Box is currently locked and set to **off.** This becomes important shortly.

3 Test the animation

- Rewind and play the animation.

  An even emission rate of 100 PPS occurs in all directions from the edit points as illustrated below:



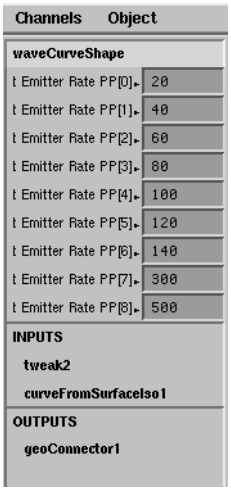
An even Omni emission from the edge curve's edit points

4 Set up Per Point Emission Control

You can easily vary the emission rate for each edit point using per point emission

- Select the *perPointEmitter*.
- Select **Particles** → **Per-Point Emission Rates**.

  The **useRatePP** attribute of *perPointEmitter* gets automatically unlocked and set to **on** in the Channel Box.
- Select *waveCurveShape* and change the emitter **Rate** for each edit point in the Channel Box.

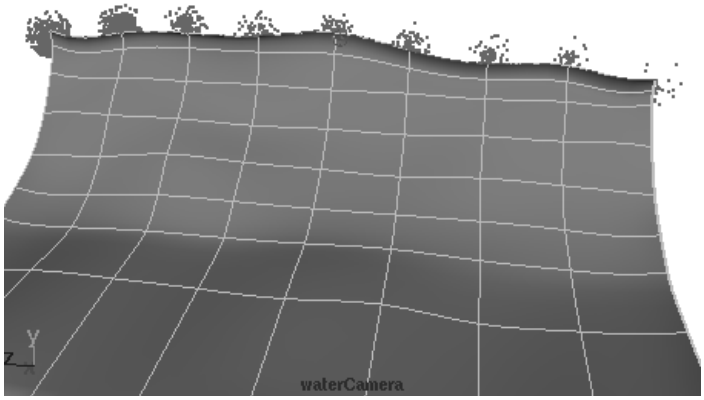


Per point emission rates shown in the Channel Box for waveCurveShape

5 Test the animation

- Rewind and play the animation.

  The emission rates of each edit point correspond to the changes made in the Channel Box. This style of emission can be combined with the curve emitter to add some variation to the overall effect.



Per point emission-rates decreasing from left to right

CONCLUSION

There are numerous ways to create particles in Maya. In this lesson you have looked at the following:

- Particle Tool
- Emitters
  - Directional
  - Omni-Directional
  - Surface
  - Texture Emission
  - Per Point Emission
- Per particle and per object attributes
- Controlling particle motion using ramps (ramp acceleration)
- Emitting into multiple particle objects

There are some additional methods for creating particles in Maya. We will be discussing those in upcoming chapters.

In this lesson you also worked on controlling the behavior of particles. Using **particle object** and **per particle** attributes such as:

- lifespan and lifespanPP
- rgb and rgbPP
- radius and radiusPP
- acceleration
- rate and ratePP

There are many more attributes that are special to particles. These will be discussed in greater detail later, however, they all are accessed and manipulated in the same way.

Exercises:

- Load *fountainGeo.ma* add curve emitters to the bowls so they drip water
- Add opacity to the bubbles in the glass example. Make the bubbles fade out just before reaching the top of the glass. Also try adding radiusPP so each particle grows slightly as it approaches the top of the glass.
- Try adding a second particle object to the bubbles emitted from the base of the glass. Use the same techniques you used with the second particle object in the fountain example.
- Combine a curve emitter and per point emission to make a wave that has more emission on end than on the other.
- Add a second particle object to the texture emission example so the texture emits multiPoint and multi-Streak.

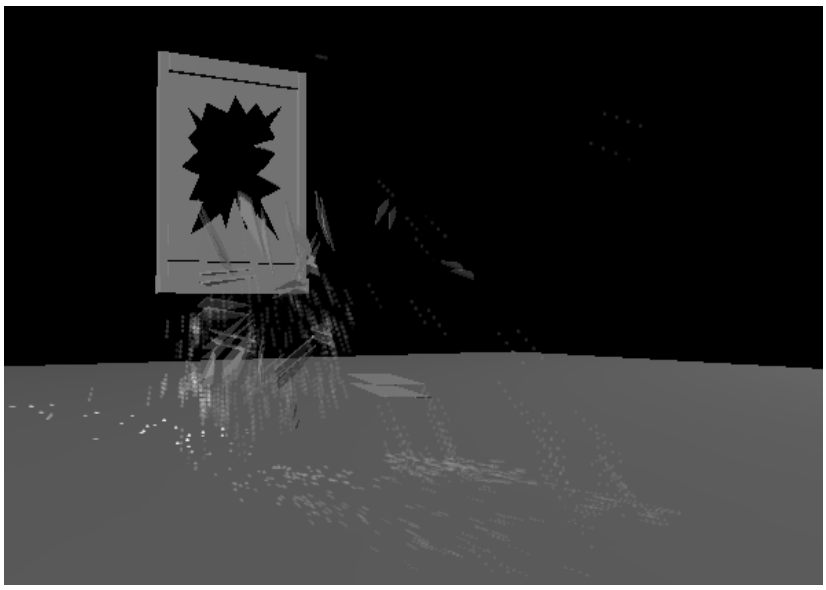
Chapter 5

5 RigidBodies & Particles

In this lesson you will combine rigid body dynamics with particles. It is common for a rigid body event such as collisions to result in a particle-involved dynamic event. Getting the two to work together can greatly assist in creating realistic movement that otherwise would require intensive keyframing.

This chapter focuses on the following topics:

- Particle and rigid body collisions
- Rigid Body optimization using collision layers
- Rigid body surface emission



.

Example: Breaking Window

In this example you are going to create a shattering window. You will be incorporating some of the tools that we have been working with up to this point as well as some more advanced techniques.

1 Open the scene

- Open the scene called *breakingGlass.mb.*

  This scene consists of a window and a floor. The window glass is made up of 3 elements whose visibility is keyed at different times so the window first appears unbroken then broken.

  glass - Unbroken pane of glass (initially visible, then keyed invisible)

  outsideGlass - Full pane after break (initially invisible then keyed visible)

  glassShards - Broken pieces that you will animate (initially invisible then keyed visible)

2 Make the Glass Shards Into Active Rigid Bodies

In order for the shards to dynamically shatter and be influenced by fields, they need to be made into active rigid bodies.

- Open the *glassShards* group and **select** all the individual *shards.*
- Select **Bodies** → **Create Active Rigid Body .**
- Edit the attributes in the Channel Box on the rigid body nodes as follows. Make sure you see "**...**" near the top of the Channel Box indicating that these edits will occur for all selected items.

  Initial Velocity Z to **1.5**
  Initial Spin Y to **5**
  Mass to **0.5**
  Bounciness to **0**
  Damping to **1**
  Dynamic and Static Friction to **0**
  Stand-In to **None**
  ApplyForcesAt to **verticiesOrCvs**

3 Assign Alternate Shards to Separate Collision Layers

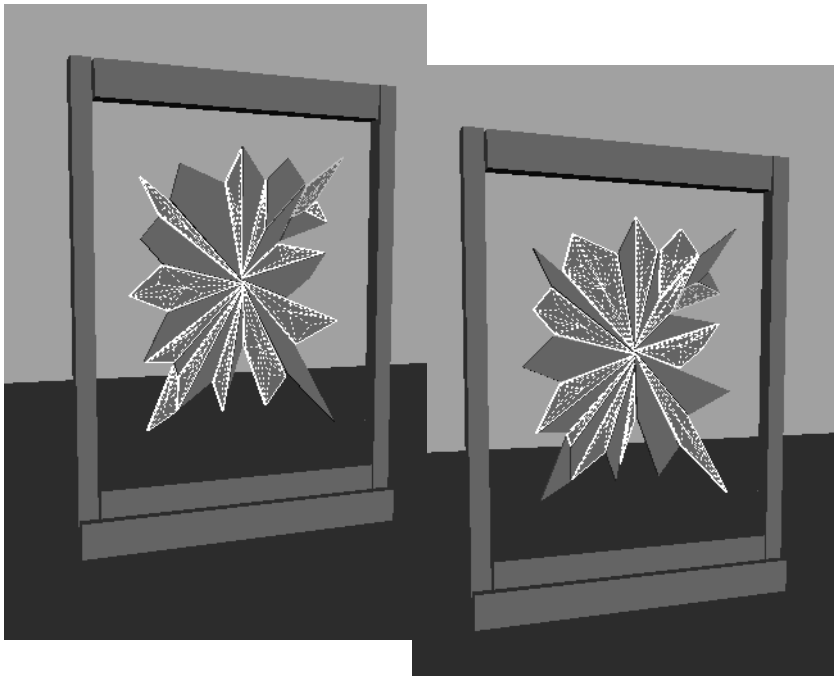
Collision layers are used to organize collections of rigid bodies into layers that the rigid body solver can handles independently.

The idea here is to put adjacent shards on separate collision layers. This will prevent excessive collision and interpenetration of neighboring objects that are initially in contact. This will also increase the performance of the solver as it only needs to determine if collisions are occurring between half as many objects.

- Select all the shard *rigidBody* nodes.
- Open **Window** → **General Editors** → **Attribute SpreadSheet...**

- In the Attribute SpreadSheet switch to the **Shape Keyable** tab then enter alternating values of **1** or **2** for the **Collision Layer** attribute as you go down the list of selected rigid body objects.

Tip: You can also key the **collisions** attribute on or off for adjacent rigid bodies. This puts each object on its own collision layer so that they do not collide with one another.



Selection for collision layer assignment

4 Make the floor a Passive Rigid Body

Now you will make the shards collide with the floor

- Select *floor*.
- Select **Bodies** → **Create Passive Rigid Body.**
- Set **Collision Layer** to **-1** in the Channel Box.

  Setting collision layer to **-1** causes the object to collide with all other rigid body objects, regardless of their collision layer setting.

5 Create Air Fields to blow the shards

- Select all the *shard* rigid bodies.

- Select **Fields** → **Create Air** –❐. and set the following:

  Magnitude to **25;**

  Attenuation to **1**;

  Max Distance to **30**;

  Direction to **0, 0, 1**;

  Speed to **0.5**;

  Inherit Velocity to 1;

  Inherit Rotation to **Off**;

  Component Only to **Off**;

  Spread to **0.3**;

- Press **Create** then **Close**
- Position the Air field behind the window facing out through the window in the Z direction.

6 Create Gravity Fields for the shards

- With the rigid bodies still selected select **Fields** → **Create Gravity**
- Set **Magnitude** to **12** in the Channel Box

7 Test the animation

The glass shards should fly out of the window and collide with the floor. Notice that only shards on the same collision layer collide with each other.

One interesting way to tune this type of animation is to get a reasonable simulation then go back to individual objects and change things like their **mass**. By creating individual mass values for each object you can improve the natural spread and interaction.

If you find, for instance, that two pieces are interpenetrating noticeably you can select one and change its mass or even its collision layer to fix the problem.

Make some pieces of glass surface emitters

1 Create Surface Emitters to throw out some Glass

Since rigid bodies can also be surface emitters, you will select 4 or 5 shards to act as surface emitters for a couple of frames.

- Select 4 evenly spaced shards.
- Select **Particles** → **Add Emitter -** ❐, and set the following.

  Emitter type to **Surface**

  This creates one emitter for each selected shard and parents the emitter to the shard.

2 Set keyframes for the rate

- Go to frame **1** and keyframe the **Rate** to a value of **1000.**
- Go to frame **10** and keyframe the **Rate** to a value of **0.**

3 Connect the Particles to the Air and Gravity fields

- **Shift-select** *particle1* then *gravityField1*.
- Select **Connect/Add** → **Connect to Field.**
- **Ctrl-select** *particle1* then *airField1* in the Outliner.
- Select **Connect/Add** → **Connect to Field.**

4 Set floor to collide with the particles

- **Cntrl-select** *particle1* and *floor* in the Outliner.
- Select **Particles** → **Make Collide.**

5 Increase Particle Inherit Factor

Increasing the inherit factor will cause the motion of the particles to inherit the speed and direction of the shards they are emitted from.

- Select *particle1*.
- In the Channel Box increase the **Inherit Factor** to **5.**

6 Playback and tune

Spend some time adjusting the various rigid body and / or field attributes to change the motion to your liking. You can also try incorporating some of what you learned in previous lessons to add more interesting characteristics to the particles such as color or opacity.

CONCLUSION

You should now have a good idea how to set up work with collision layers, basic particle collisions, and methods for integrating particles with rigid bodies. One of the big challenges with rigid bodies is handling objects that are stacked together. You've seen that collision layers are useful for handling this situation and also provide a method for speeding up the solver evaluation.

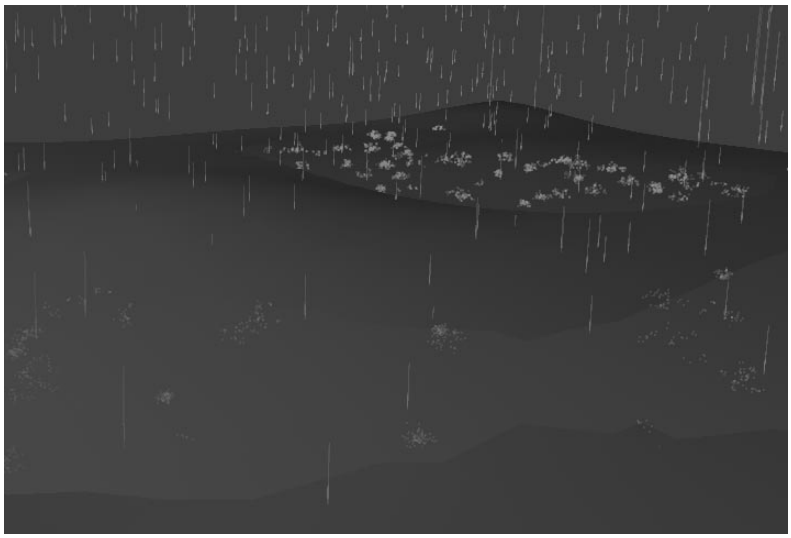
In upcoming lessons, you will take this a step further to work with particle collision events which cause a user defined action to occur when a particle collision occurs.

Chapter 6

6 Particle Collision

In this lesson you will learn how to create and tune Particle Collisions:

- Creating Particle to Surface Collisions
- Creating and Editing Particle Collision Events
- Particle Collision Event Procedures
- Applications



CREATING PARTICLE TO SURFACE COLLISIONS

To enable a particle to collide and interact with a geometric object (including soft bodies, trimmed objects and deforming geometry) you will use the **Particles** → **Make Collide** menu command.

Particles can collide with geometry but cannot collide with each other.

Particle Collision Events

With the Particle Collision Event you can trigger the following events when a particle collides with a collision object:

- Split a particle into other particles
- Emit new particles
- Execute a MEL procedure
- Kill particles

The collision can be caused by either moving particles or by moving or deforming geometry.

Collision Events obey trimmed surfaces.

Exercise: Rain Drops

This exercise will introduce you to the Collision Event Editor.

1 Open rainDrops.mb

- Open *rainDrops.mb*

  This scene file contains three pieces of geometry :

  rainCloud - polygonal plane

  rainSurface - NURBs surface obtained from trim

  mounds - NURBs surface

2 Add a surface emitter to the rainCloud object

- Select the *rainCloud* plane.
- Select **Particles** → **Add Emitter -** ❐, and set the following:

  Emitter Type to **Surface**
- Press **Create.**
- Set the emitter's **rate** to **1** in the Channel Box.

3 Set attributes for particleShape1

- Set the following attributes for *particleShape1*:

  Particle Render Type to **Multi-Streak**

  Line Width to **1**

  MultiCount to **3**

  Multi Radius to **.01**

  Normal Dir to **2**

  Tail Fade to **0.5**

  Tail Size to **0.5**

4 Make rainSurface a collidable object

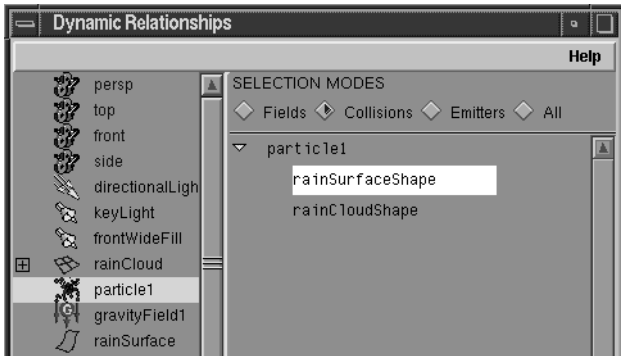
- Select *rainSurface*.
- Select **Particles** → **Make Collide.**

  This will create and connect a *geoConnector* node to the *rainSurface* object. This node contains the collision attributes of **Tessellation Factor**, **Resillience,** and **Friction**.

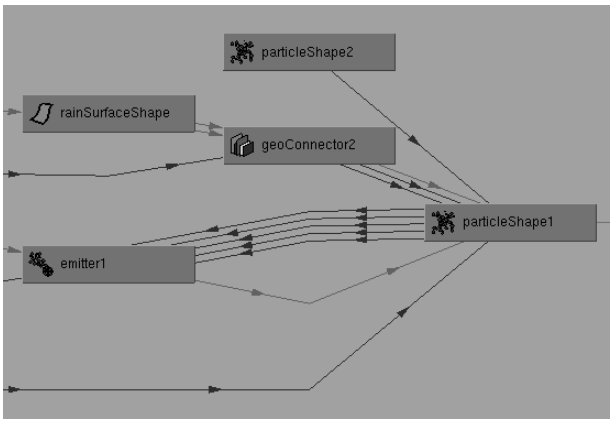
  The *rainSurface* geometry will now show up as an option for connecting the particle object as a collision in the Dynamics Relationship Editor.

5 Connect rainSurface to particle1

- Open the Dynamic Relationships Editor.
- Select the *particle1* object.
- Select **Collisions** under **Selection Modes.**
- Select *rainSurfaceShape.*



The connection is made between the *rainSurfaceShape* and the *particleShape1* nodes via the geoConnector node.



6 Test the scene

Test the scene for proper playback. You should see the particles now colliding with *rainSurface*.

Adjust the **resillience** and **friction** as desired.

Tip: You can also do this collision and connection in one step by first selecting the *particle* then the *surface* it is to collide with then press **Particles** → **Make Collide**.

---

7 Use the Particle Tool to Create an empty particle object

You will soon make the colliding particles to split into new particles. Creating an empty particle object now will give those new particles somewhere to be stored.

- Use the Particle Tool to Create an empty particle object.

  When created, the empty particle object is called *particle2*.

8 Connect Particle2 to rainSurface

You can set up collision and connection in one step without using the Dynamic Relationship Editor.

- **Shif- select** *particle2* then *rainSurface* object.
- Select **Particles** → **Make Collide**.

Using the Particle Collision Event Editor

1 Open Collision Event Editor

- **Particles** → **Particle Collision Events...**

  This editor is the graphical interface to the `event` MEL command.

Selection and Events Section

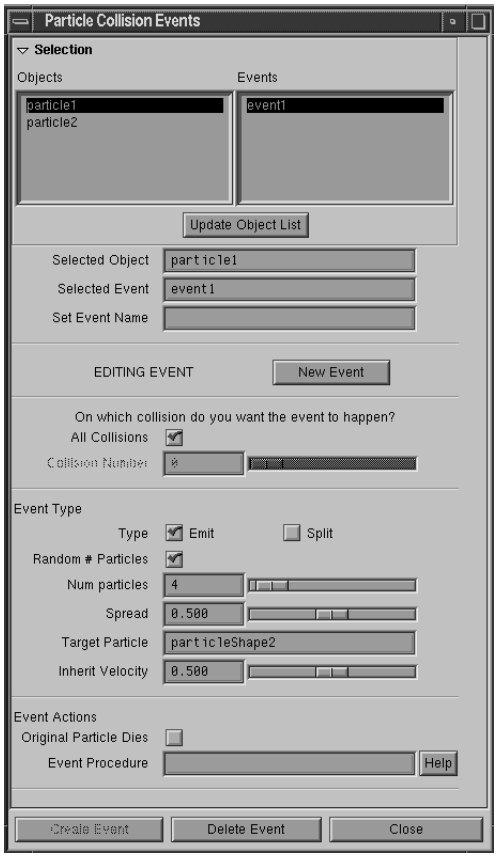
The top two list panes provide for selection of valid objects to create events for and for selecting events to edit.

The next section provides fields that give information about the selected event and allow for editing the event name.

You can create multiple events for each particle you have selected in the lefthand list. You can also update the list of particle objects by pressing the **Update Object List** button.

Below that section is a section which displays whether you are in edit or creation mode and a button to add a New Event to the particle you are working with.

The next section lets you choose which collision the event occurs on or all collisions.



From this editor you can choose actions to occur when a particle collides with its collision objects.

2 Set Event options

- Select *particle1* in the Objects section.
- Set the following options as directed:

  All Collisions to **On;**

  Collisions Number to **N/A**

  Type to **Split;**

  Num particles to **20;**

  Spread to **1;**

  Target Particle to **Particle2Shape;**

  Inherit Velocity to **.8**

3 Create Event

- Press **Create Event**

4 Test the animation

Playback the simulation. Note a new particle was created for you.

Event Type Section

Emit vrs Split

Whether the particle will Emit new particles and live "**Emit**" or Emit new particles and Die "**Split**". If a new target particle is not specified for Split then it uses the same particle object as the colliding particle.

Random # Particles

Checking this attribute will create a random amount of emitted particles with a min range of 0 and a max range of the following attribute Num Particles.

Num Particles

Sets the amount of particles emitted at collision or the max range of particles if Random # Particles is selected.

Spread

Controls the spread of the emitted particles. Valid values are from 0 to 1.

Target Particle

The Target Particle field is where you can choose the particle Shape that you want to emit upon the collision event. If you do not select a particle one is created for you.

Inherit Velocity

This value controls what percentage of the parent particles velocity will be assumed by the new Split or Emitted particles.

The Event Actions Section

The Event Actions Section is where you can call a mel procedure at the time of collision. Later in this lesson we will look at this section more closely.

Tip: You can also press **Create Event** first, then go back and edit the event settings.

5 Add two Events to the particle2

- Open **Particles** → **Particle Collision Events...**
- Select *particle2* in the object list pane.
- Press **Create Event.**
- Rename your first event from *event1* to *firstEvent* in the **Set Event Name** field.
- Press **New Event**.

  You are now in create mode
- Press **Create Event** at the bottom of the window
- You have now created another new event for *particle2* called *event2.*
- rename this event to *secondEvent.*

6 Set Event Options for firstEvent

Set the following options as directed:

All Collisions to **off;**

Collision Number to **1;**

Type to **Split;**

Num Particles to **2;**

Spread to **.5;**

Target Particle to **particleShape2;**

This is the new particle that you created

Inherit Velocity to **.5**

7 Set Event Options for secondEvent

Set the following options as directed:

All Collisions to **Off;**

Collision Number to **2;**

Type to **Split;**

Num Particles to **0;**

Spread to **.5;**

Target Particle to

Inherit Velocity to **.5;**

Original Particle Dies to **On**.

8 Create Gravity on particle2

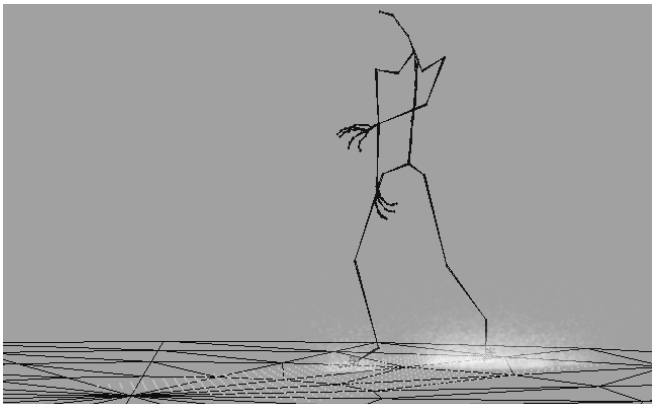
- Select *particle2***.**

- Select **Fields** → **Create Gravity.**

9 Run the scene and tune

Example: Trampled under Foot

The scene file *footDust.mb* contains a character and floor of particles. You will make the character kick up a cloud of particles as he walks through the scene.



PinHead walking through dust

1 Open the scene

- Open the file *footDust.mb*

  Note the scale of the Grid. We are working on a larger world scale. Many attributes on the dynamic objects will need to be set in relation to this larger scale.

2 Test the animation

You will see the character walk through the particles.

3 Create a radial field

- Select *floorParticles.*
- Select **Fields** → **Create Radial** - ❐, and set the following:

  Magnitude to **1;**

  Attenuation to **1;**

  Max Distance to **5;**

  Use Max Distance to **on**

4 Point Constrain the field to the left foot

- Select the left foot *Ball* joint.
- In the Outliner Select **Display** → **Selected** to see the object nested in the hierarchy.
- In the Outliner**, Cntrl-select** *Ball* then the radial field.
- Select **Constrain** → **Point.**
- Create another radial field and constrain it to the other foot

5 Test the animation

Playback the scene to ensure that the fields and particles are interacting.

6 Set the floor particles to collide with the floor object

- Select the *Particle1* object then **Shift-Select** (Hypergraph) **Cntrl-Select** (Outliner) the *stageBase* object.
- Select **Particles** → **Make Collide.**

7 Create a Collision Event for the floor particles

- Select **Particles** → **Particle Collision Event...**
- Select *particle1* in the Objects Section
- Press **Create Event**.
- Select *event1* in the Events Selection pane.
- Enter the following Event Type options:

  Type to **Emit;**

  Random # Particles to **On;**

  Num Particles to **10;**

  Spread to **1;**

  Target Particle to **particleShape2;**

  Inherit Velocity to **1;**

8 Playback and Tune

There are various places to fine-tune this example. Particle **lifeSpan** and **Max Count** will control how many particles are in the scene.

Resilience on the *stageBase* object's *geoConnector* will control how the particles react with the floor.

Particle Collision Event Procedure

You have learned how to create a collision event. Now lets take things a little farther and explore our options for triggering a more complex animation at the time of collision.

The Particle Collision Event also has a section called Event Actions. From this section you can enter an *Event Procedure.* An Event Procedure is typically a MEL script that is called when a collision occurs and the event is triggered. There are a multitude of applications that can utilize this functionality.

There is one requirement for the script that is called by the Particle Collision Event. It must have the following format and argument list:

```
global proc myEventProc(string $particleName, int
$particleId, string $objectName)
```

Where `myEventProc` is the name of the MEL procedure and also the name of the script (myEventProc.mel), `$particleName` is the name of the particle object that owns the event, `$particleId` is the particle number of the particle that has collided, and `$objectName` is the name of the object that the particle has collided with.

These arguments which are also variables are the placeholders for holding the information that is passed to the script from the Particle Collision Event.

1 Open the scene

This scene has been prepared to accept a specific Particle Collision Event procedure. It contains the rain objects you used before and also a *closestPointOnSurface* node that has been connected to the ground object.

- Open the scene *collisionScript.mb*

These are the commands that were used to create this node and the connection to the *ground* surface:

```
createNode closestPointOnSurface;
connectAttr -f ground.worldSpace[0]
     closestPointOnSurface1.inputSurface;
```

The *closestPointOnSurface* node is a very useful node for querying the world and UV position of a point on a surface.

2 Set Particle1 to collide with ground

- **Shift-Select** *particle1* and then *ground.*
- Select **Particles** → **Make Collide.**

3 Add the script procedure to a Particle Collision Event Action Event

- Select **Particles** → **Particle Collision Event...**
- Press **Create Event.**

Note: Do not select Emit or Split as the Event Type

- Enter **partCollisionPrnt** into the Event Procedure field under the Event Actions section.

4 Open the script editor and playback the scene

The *partCollisionPrnt* script is executed each time a particle collides with the surface. The *partCollisionPrnt* script then prints the position on the surface that the collision occured.

Sample output from the script editor:

```
partCollisionPrnt( "particleShape1" , 0, " ground"  );
CPOS XYZ        10.51686562 2.943025257e-15 -10.38921561
CPOS UV         0.9370036721 0.9316994754
POS Position    10.51686562 2.943025257e-15 -10.38921561
POS UV Position 0.9370036721 0.9316994754
```

5 Open the partCollisionPrnt.mel script

The procedure *partCollisionPrnt* does a few things. First is takes the arguments given to it from the Particle Collision Event and puts these values into global variables:

```
$particlePositions
$particleVelocitys
$hitTimes
$particleHitCount
$currentHitTime
```

These variables can then be accessed from other procedures or expressions in Maya. Use this upper portion of the script as a template for your own Particle Collision Event scripts.

The second portion uses two types of point on surface nodes to get and maintain collision information as it pertains to the surface.

closestPointOnSurface returns information about a point on the surface in relation to the worldspace position information that the $particlePositions variable is getting from the Particle Collision Event each time a particle collides with the surface.

pointOnSurface is an operation that can create a *pointOnSurfaceInfo* node. This node will maintain information about a point on a surface even if the surface is animating and deforming.

CONCLUSION

The Particle Collision Event is a very powerful method of controlling particles and their behavior. It provides a logical method of emission and or death based on collision. The ability to execute a procedure at collision also opens up a large range of possibilities for creating geometry or manipulating virtually any other part of maya or even you system at times of collision.

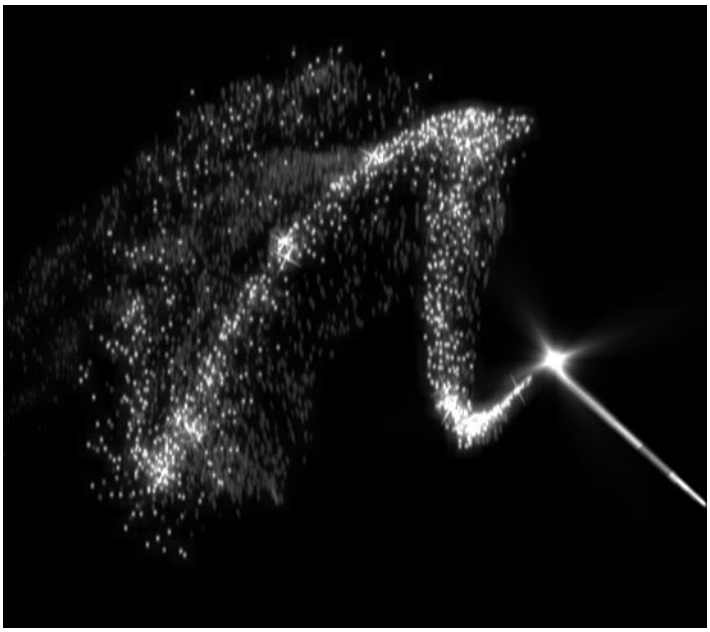
7 Particle Expressions

CONTROLLING PARTICLE MOTION

This lesson focuses on different techniques for controlling particle motion with special attention placed on particle expressions.

The following topics will be discussed in this lesson:

- Fundamental physics concepts
- Maya's particle evaluation process
- Initial state
- Creation vs. runtime expressions
- Linstep and smoothstep functions
- The particleId attribute
- Absolute Value
- Sin and Cos



Chapter 7

Fundamental physics concepts

There are some basic rules that govern the motion of objects in the Universe that are directly applicable to a discussion of particles in Maya. Newton's first law states the following:

```
Force = Mass * Acceleration
```

This is more commonly written as: `F = ma`.

Force and **mass** are known quantities in Maya. Acceleration is calculated by the Dynamics system based on these values. The resulting values are used to control the particle's motion.

Force is a generated quantity that can come from things like *fields*, *springs* and *expressions*.

Mass is an attribute that exists by default on particle objects. Therefore, since two items in the equation are known, the third (**acceleration**)can be determined through simple division.

```
a= F/m
```

This rule forms the basis of the underlying architecture Maya uses to calculate particle attributes such as position, velocity, and acceleration. Understanding this relationship is not always necessary but can be useful when deciding how to set something up, or when troubleshooting.

Some useful definitions:

There are some common terms and definitions that come up frequently regarding particle attributes and quantities related to particles.

Scalar: A numerical quantity with only 1 specific component. *Time,* and *mass* are examples scalar values represented by values like 20, or -3.5.

Vector: A quantity with magnitude and direction. In Maya, this is represented as 3 distinct numerical components grouped together in brackets i.e. <<1,2,3>> or <<5,-2,1>>.

Float: A decimal numerical value i.e. 2.3, 0.001, 3.14, etc.

Integer: A non-decimal whole number i.e. -1, 0, 57, etc.

String: A collection of alphanumeric characters i.e."hello"

Boolean: A value that is either *true* or *false*, *on* or *off*, *1* or 0.

Position: (vector): A particle's *location in the world* is its **position**.

Velocity: (vector): **Velocity** is the *change in position over time*. This is a measurement of both **rate** and **direction**. To visualize velocity, imagine an arrow pointing in the direction of the object's motion with the arrow's length proportional to the speed of the object.

Speed: (scalar): Speed differs from velocity in that it is a measurement of *rate only* (without respect to direction).

Acceleration: (vector): Acceleration is a measurement of the *change in velocity* over time.

Propagation: the evaluation process

Propagation is the method Maya uses to determine a particle's attribute values by basing the calculations for the current frame on the result that was determined from the previous frame.

Propagation is kind of like a "piggy-back" effect. For example, frame 2 gets information from frame 1, does some calculations, then positions the particles. Next, frame 3 gets the result from frame 2, does calculations on that, positions the particles, and moves to frame 4. The cycle continues throughout the playback of the animation.

So...what happens at frame 1?

Frame 1 in the above example is the **initial state** of the system. **Initial state** refers to the values that exist in any dynamic object's attributes at the initial frame of a dynamic simulation. It is from this initial state that propagation occurs.

Note: The **initial state** of a simulation is not necessarily the first frame in the playback frame range but, instead, is determined by the **start frame** attribute on each particle object.

Creation vs. Runtime expressions

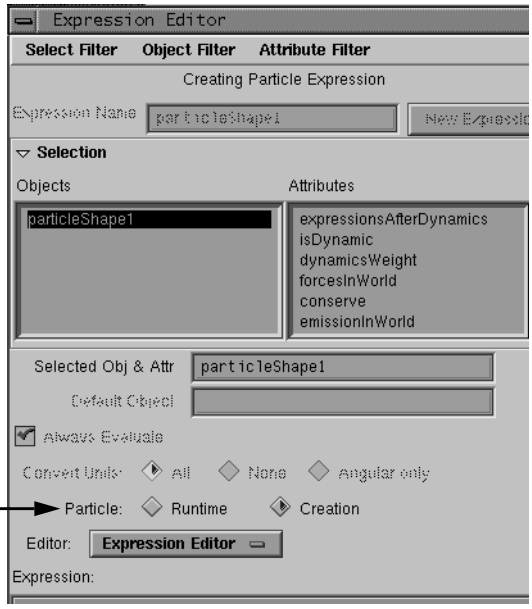
It is important to understand the difference between creation and runtime expressions.

Creation Expression: Evaluated only once for each particle when the particle is born.

Runtime Expression: Evaluated at least once per particle per frame but not at particle birth.

Note: Particle expressions are also commonly referred to as "**rules**." The term **rule** comes from **Dynamation** which uses **creation** and **runtime rules**.

- Each particle object stores all of its expressions in one of two places: the **creation expression** or the **runtime expression**.
- All creation expressions are stored in the creation portion of the Expression Editor for that particle object.
- Likewise, all runtime expressions on a given particle object reside in the runtime segment of that particle object.



Toggle between Runtime and Creation

The Expression Editor can toggle between displaying the runtime and creation expressions for the selected particleShape node. The expressions are evaluated in the order they appear in the Expression Editor.

Tip: Runtime and Creation will be grayed out unless the particleShape node is the selected item.

---

1 Create a simple creation expression

- Open the file *runtimeCreation.mb*
- Switch to shaded mode and playback.

  This scene contains two pre-made directional emitters with an rgbPP attribute already added to both particleShape nodes. Right now, the rgbPP value is <<0,0,0>> so the particles emit as black.
- Open the Attribute Editor for *particleShape1*.
- **RMB** on **rgbPP** and select **Creation Expression.**
- Enter the following expression in the Expression Editor:

```
rgbPP = <<rand(1),0,0>>;
```

Note: It is common to see many different attributes assigned values on the **same line** in the Expression Editor, each should be separated by semi-colons. i.e. `rgbPP = <<1,0,1>>; lifespanPP = rand (4,6);`

---

2 Create a simple runtime expression

- Select *particleShape2*.
- **RMB** on **rgbPP** and select **Runtime Expression**.

- Enter the following expression in the Expression Editor:

```
rgbPP = <<0,0,rand(1)>>;
```

3 Test the results

- Make sure you are in shaded mode.
- Rewind and then playback

  The particle shape with the creation expression gets a random red color assigned to it only once during the animation. The particle shape with the runtime expression reassigns a new random blue value on each frame of the animation.

Note: The syntax **rand(1)** picks a random value between 0 and 1. You can also define a more specific range by using two numbers with rand. For example **rand (20, 30)** picks a random value between 20 and 30. This example uses a range from 0 to 1 since RGB values range from 0 to 1.

Use the same techniques to create additional creation and runtime expressions to control other attributes such as radiusPP, lifespanPP, or opacityPP.

APPLIED PARTICLE EXPRESSIONS

You should now have a better understanding of the difference between runtime and creation. The next step is to use these concepts in conjunction with normalized age to establish a relationship between time and the attribute values.

The following template can be used when writing a particle expression to animate from **variable X** to **variable Y** over the particles **age**:

```
A+ (B-A) * (age/lifespan)
```

1 Mimic a ramp behavior with a particle expression

In a previous lesson, you learned to change the color of a particle over its age based on a ramp. This section teaches you how to do the same thing using an expression. This is handy if you need some specific control that you can't get from a ramp.

- Open the file *fountainExpression.mb*
- Open the Attribute Editor for *mist*.
- Add a **per object lifespan** attribute and set it to **2.6**.
- Add an **rgbPP** attribute to *mist*.
- Press **RMB** in the **rgbPP** field and choose **Runtime Expression.**
- Following the template from above, substitute values for white (A),light blue(B), and normalized age into the expression editor as shown below.

```
$normAge = age/lifespan;
vector $start = <<1,1,1>>;
vector $end = <<0,0,0.8>>;
```

```
rgbPP = $start + ($end-$start) * ($normAge);
```

- Drag the contents of this expression to the shelf, you'll use it again later.
- Press **Create** then close the Expression Editor.
- Press **5** to switch to shaded mode.
- Rewind then play the animation.

  The expression causes the particles to slowly transition from white to blue over the particles' age in the same fashion ramps controlled their color in a previous lesson. You can adjust the start and end colors in the expression to your liking once you see the effect the expression is having.
- The particles at the very base of the stream where the particles leave the emitter are black. See if you can use what you've already learned to fix this.

Linstep and smoothstep

Linstep and smoothstep are built-in functions within Maya that return a value between 0 and 1 over a specified range for a given unit (frames, fps, lifespan, age, etc).

Linstep produces a linear curve, smoothstep a linear curve with an ease in and ease out appearance at the tangents.

The syntax template for a linstep (or smoothstep) statement is as follows:

```
linstep (start, end, unitParameter)
```

One advantage of using linstep and smoothstep is that the range of the effect can occur over any defined interval instead of being limited to the particle's age.

It is also possible to make the range of the values for linstep or smoothstep extend beyond the range of 0 - 1. For example, to make a particle's radius increase from 0 - 5 over the course of frames 8-20 the following runtime expression could be used:

```
radiusPP = 5* linstep (8, 20, frame)
```

To make a linstep curve decrease instead of increase, subtract the linstep statement from 1. Below is a common linstep function that will cause opacity to fade out linearly over the particle's age if placed in the runtime expression.

```
opacityPP = 1-(smoothstep(0, lifespanPP, age));
```

Note: Graphs for the linstep and smoothstep function are provided in Maya's on-line documentation

---

Taking it a step beyond ramps

So far, what you've done could be done using ramps. The idea has been to get you familiar with how to enter expressions and how their evaluation works in Maya. Now, you'll control a particle attribute such as radiusPP

using a dynamic attribute like velocity. This is more difficult to accomplish with ramps and lends itself well to an expression.

1 Set mist render type to sphere

Although you may not want to render the final shot in sphere mode, for this example you will use the sphere render type since the effects of the expression are easiest to see with that render type.

2 Double check that rgbPP expression is still present

- Look in the runtime expression for rgbPP to make sure the expression you used for rgbPP above is still assigned to the particle object. If you deleted it, drag and drop it from the shelf back into the Expression Editor.

3 Add a radiusPP attribute and enter a runtime expression

- Add a **radiusPP** attribute
- Enter the following runtime expression for radiusPP, You may want to enter a few carriage returns below the expression that is controlling rgbPP to make things more readable.

```
float $startRadius = 0.1;
float $endRadius = 0.5;
vector $vel = velocity;
float $y = $vel.y;
radiusPP = $y/10 * ($startRadius + ($endRadius-
$startRadius) * $normAge);
```

The above expression changes the radius of the spheres based on a factor of their velocity in the **y** direction but also based on their normalized age. Notice that the radius decreases when the drops slow down then increases as they speed back up.

What happened to the rgb values?

You may have noticed that now the spheres begin white but then turn black at some point instead of colored. Any ideas why this happened?

4 Use the numeric render type to track down the problem

The numeric render type is useful for determining what values a specific particle attribute holds.

- Playback so you have some emitted particles.
- Select the *mist* particles.
- Set **Render Type** to **Numeric.**
- Press **Current Render Type** and enter **radiusPP** in the Attribute Name field.

  This displays the numeric radius value held for each particle. As you playback, the values start positive then become negative. Since the expression used is returning negative radius values, the spheres get turned “inside out.” This is why you see black instead of color.

5 Correct the expression so only positive radius values are used

- Open the **Runtime Expression** for **radiusPP**
- Edit the last line of the expression so it appears as follows:

```
radiusPP = abs($y/10 * ($startRadius +
($endRadius-
$startRadius) * $normAge));
```

The only difference is that you enclosed what you previously had within **abs()**. Abs is a function that takes the **absolute value** of the value within parenthesis. This tells the expression to check the value in parenthesis, and make it positive if it is negative.

- Press **Edit** then **Close**.

6 Test the results

- Playback in **Numeric** mode again to verify that the numbers stay positive throughout.
- Switch **Render Type** back to **Sphere** then playback to see that the rgb values now display correctly.

Particle Motion Examples

The file *expressionExamples.mb* contains several particle objects in different display layers. Each particle object has its own creation and/or runtime expression illustrating a common or interesting technique used with particle expressions.

Exercise: Magic Wand

This is an application for controlling particle color over time to create the classic pixie dust effect.

1 Load the file

This scene consists of a cylinder object called wand and some standard lighting.

- Open the file *magicWand.mb*.

2 Create an emitter and parent it to the wand geometry

- Create a directional emitter with default values.
- Rename emitter *dustEmitter*.
- In the Outliner **MMB-drag** *dustEmitter* onto *wand*.
- Select *dustEmitter* and translate it to the end of the *wand* geometry.

3 Add an emitter to the dust particle object

- Rename *particle1* to *dust*.
- Select *dust*.
- Select **Particles → Add Emitter.**

  This creates an emitter that will emit particles from the dust particles. It also creates another particle object.
- Rename the added emitter *trailEmitter*.
- Rename the new particle object *dustTrail*.

4 Adjust Emitter attributes

- For the *dustEmitter* set the following:

  Emitter Type to **Directional**

  Rate to **800**

  Spread to **0.5**

  Speed to **2**
- For the *trailEmitter* set the following:

  Emitter Type to **Directional**

  Rate to **1**

  Spread to **0.2**

  Speed to **0**

5 Create fields for the pixie dust

- Add **gravity** to *dust* and increase **Magnitude** to **12**
- Add a separate **gravity** field to *dustTrail* with **Magnitude** to **4** and **Attenuation** to **0**.
- Add **Turbulence** to *dust. S*et the **Magnitude** to **5**

6 Adjust dust particle shape attributes

Render Type to **Points**

Normal Dir to **2**

Point Size to **2**

7 Adjust dustTrail particle shape attributes

Render Type to **Streak**

Line Width to **1**

Normal Dir to **2**

Tail Fade to **1**

Tail Size to **.05**

8 Add Per Particle (Array) Attributes to the dust particle shape

- Select *dust*.
- Add a **lifespanPP** attribute
- Add a **rgbPP** attribute

9 Add Per Particle (Array) Attributes to the dustTrail particle shape

- Add a **lifespanPP** attribute
- Add a **rgbPP** attribute
- Add a **opacityPP** attribute

10 Create a Creation Expression for the lifespanPP of the dust particle shape

- Use the following creation expression to control the dust particle lifespan on a per particle basis:

  ```
  lifespanPP = rand(1,3);
  ```

  This expression assigns a random lifespan value between 1 and 3 seconds to each particle.

11 Create a Runtime Expression for the rgbPP of the dust particle shape

- Use the following classic twinkle expression to control the particle color on a per particle basis:

  ```
  rgbPP = <<1,1,1>> * (sin(0.5*id + time * 20));
  ```

  This above expression controls the color of the particles on a per particle basis. Here is a breakdown of what is going on:

  <<1,1,1>> - This is the rgb vector value of white. The expression multiplies a number against this value to change its overall value by <<0,0,0>> (black) and <<1,1,1>> (white).

  sin(.5 * id + time * 20) sin is a function that can create an oscillating value between 1 and -1.

  By multiplying sin by variables like `particleId` and `time` we can get values that are unique and changing rhythmically. This is a very important function of expressions...especially particle expressions.

  0.5 * id - when working with per particle expressions it is useful to work with the `particleShape.particleId` attribute. This attribute as you have seen gives us a unique value for each particle that this expression is applied to.

0.5 * id + time * 20 - Again time is a great incrementer. Multiplying by 20 in this case dictates how fast or the frequency of this sin functions repetition.

Why does this expression work?

What is strange about this expression?

Alternate Expression:

Here is an alternate expression that does not use negative values against the rgb vector:

```
rgbPP = <<1,1,1>> * ((sin(0.5 * id + time * 20)
*0.5)+0.5);
```

This example offsets the sin function to provide values that fall between 0 and 1. To do this the sin is multiplied by 0.5 to cut the amplitude in half. An offset has also been added to keep its values above 0.

Will this work better or worse?

12 Make a Creation Expression for the lifespanPP

- Select *dustTrail* particle shape.
- Add the following Creation Expression:

```
lifespanPP = rand(2,5);
```

For the dustTrail we want these particles to live a little longer.

13 Create Runtime Expressions for the rgbPP and opacityPP of the dustTrail particle shape

You will create the same type of **rgbPP** expression and also an **opacityPP** expression to control, not only the **color**, but also the **transparency** on the particles.

```
rgbPP = <<1,1,1>> * ((sin(.5 * id + time * 20)
*.5)+.5);
```

```
opacityPP=(1-((linstep(0,lifespanPP, age))) *.0005);
```

1-linstep(0,lifespanPP,age) - The linstep function is used here to provide a linear change of value between 0 and 1 over time. 1-linstep gives us the reverse, returning values from 1 to 0 over the particle's age. This value is different for each particle based on the lifespanPP creation expression you already made.

14 Playback and tune

- Experiment with field attributes, render types or multiplier values in the expressions to tune the results.

Exercises:

- Simulate the effect of gravity using a particle expression instead of a field.
- Create a particle expression that assigns a random radius value between 1 and 4 to a particle as it is emitted.

- Create a particle expression that changes each particle's color from yellow to blue over its age.
- Create a particle expression using a dynamic attribute such as velocity, acceleration, position, or mass and rand, sphrand, noise or dnoise.

CONCLUSION

You now have a foundation for creating some of your own particle expressions. Particle expressions should be considered as another tool within Maya's dynamics. Not all situations lend themselves well to using expressions. However, they give you access to a lower level of information that, in some cases, is not accessible through graphical methods such as ramps or the Attribute Editor. Expressions can also provide a solution for getting results that would be difficult or impossible to keyframe and can give your simulations the ability to have decision making built into them.

Chapter 8

8 Flow

Flow is a built-in effect that allows you to quickly and easily make particles follow the shape of a specified curve. In this chapter, you will create a torondo effect using flow.

Use flow to create a basic tornado effect

1 Load a file

- Select **File** → **Open.**
- Open *tornadoCurve.mb*

2 Apply a lattice deformer

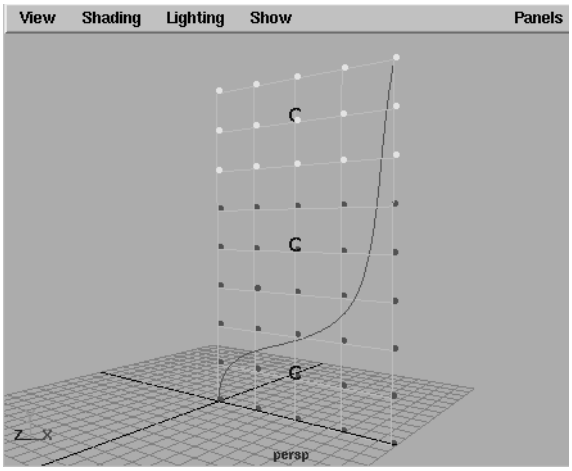
- Select *tornadoCurve***.**
- Select **Deform** → **Create Lattice -** ❐, and set the following:

  Divisions to **5, 9, 2;**

  Parenting to **on**
- Press **Create**.

3 Add relative clusters

- Select the *ffdLattice* node then press **F8** to switch to component mode.
- Select the top three rows of lattice points.



Adding Clusters to the Lattice Points

- Choose **Deform -→ Create Cluster -** ❐, and make sure **Relative Mode** is **on.**
- Press **Create**.
- Make relative clusters for the middle 3 rows of lattice points
- Make relative clusters for the bottom 3 rows of lattice points

---

Note: Making a cluster **relative** forces the cluster to receive its transform information from the transform node directly above it in the hierarchy. This prevents the *double transformations* that normally occur when clusters are parented to geometry. It is possible to toggle the relative mode on and off in the Attribute Editor on a cluster that has already been created.

---

4 Group the clusters under the tornadoCurve

- Use the **MMB** in the Outliner to parent the 3 cluster handles to *tornadoCurve***.**

5 Keyframe the lattice/curve

- Select *tornadoCurve.*
- Rewind to frame **1;**
- Move *tornadoCurve* to **-12, 0, 12.**
- Press **Shift+W** to keyframe the translation channels.
- Advance to frame **110** and set another keyframe using **Shift+W.**
- Advance to frame **160.**
- Move *tornadoCurve* to **0 0 0** and set another keyframe.
- Advance to frame **300.**
- Position the *tornadoCurve* at **-12, 0, -12** and set another keyframe.

  You can also add keyframes to the cluster handles if you wish to animate the shape of the funnel instead of only the position.

  It is best to get the motion of the curve the way you want it before animating the clusters.

6 Add flow to the curve

- Select *tornadoCurveShape.*
- Select **Effects** → **Flow -** ❐, and set the following:

  Flow Group Name to **tornadoFlow;**

  Control Segments to **6;**

  Particle Lifespan to **3 ;**

  Goal Weight to **0.62;**
- Press **Create.**

  A new node called *tornadoCurveShape_flowGroup* is created. This node contains all the flow compoents Maya needs to control the flow of particles along the curve.

Note: Deformers can be added to a flow path curve after flow has already been applied. Flow does not interrupt the curve history.

---

7 Test the results

- Rewind then playback the animation

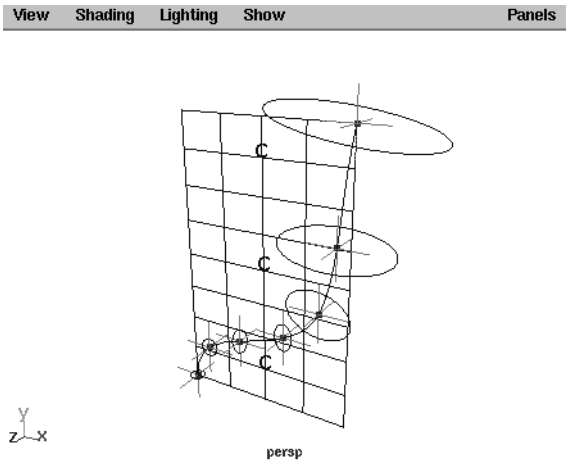
  The particles follow the shape of the curve as they are emitted. They reach the end of the curve in 3 seconds (90 frames) since **lifespan** is set to **3**.

  The particles do not flow exactly along the path of the curve. This is due to a low **Goal Weight** setting. Goal weight values closer to **1** cause the particles to adhere more closely to the curve's shape.

If you want to adjust these values, select the *tornadoCurveShape_flowGroup* node.

8 Adjust flow parameters

- Select the circle (control segment) at the upper end of *tornadoCurve* and scale it up.
- Scale the other circles along the curve in a similar fashion so they increase in scale from the bottom of the curve to the top as shown below :



Flow circles from small to large

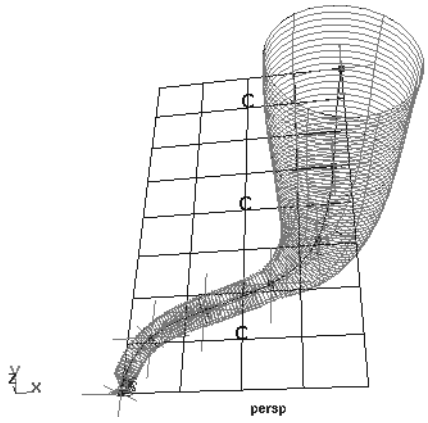
Tip: To adjust the control segments (rings) along the length of the curve pick the selection handle (or the flow group in the outliner) at the beginning of the curve then edit the attributes called Locator_xx_pos in the Channel Box. (xx corresponds to the number of the control segment)

9 Enable Display Thickness

You can get a better sense of the volume the particles will fill by enabling the display thickness attribute for refining your circles

- Select *tornadoCurveShape_flowGroup.*
- Set **Display Thickness** to **On** in the Channel Box

  There are additional display attributes such as Display Subcircles and Display All Sections. These values are used for advanced refinement of the curve and to help smooth out sharp bends in the flow path if necessary.
- Once satisfied with the shape, set all display attribes back to **off**



Flow Curve With Display Thickness Enabled

10 Add and adjust a vortex field

- Select the particles then choose **Fields** → **Create Vortex -** ❐, and set the following:

  Magnitude to **20**

  Attenuation to **0**

- Select the vortex field and position at the base of *tornadoCurve.*
- Use the Outliner to parent the vortex field to *tornadoCurve*

11 Add a ramp to control radius over age

- Switch the particles' **Render Type** to **Cloud.**
- Add a **radiusPP** attribute.
- **RMB** to create a ramp on **radiusPP** .
- Adust the **R** value of the ramp so the **radius** starts at **0.1** and die at **0.9.**

Tip: The radius could also be increased over age using the runtime expression : `radiusPP = 0.1 + ((0.9-0.1) * smoothstep(0,lifespan,age));`

Since the goal weight is fairly low,the vortex field adds to the acceleration of the particles and allows them to spin as they flow. As the goal weight value is increased, the effect from the vortex is less but the particles adhere more closesly to the curve.

When combining fields with flow effects, the curve usually ends up acting as a gernal guide for the particles --not an absolute pathway for them. This allows the field to influence the particle.

Conclusion

The Flow Clip Effect is a handy method for getting particles to go where you want them to go. It often will provide much more control then very elaborate expressions.

Use Flow for things like:

- Water flowing
- Energy Streams
- Flocking
- Smoke

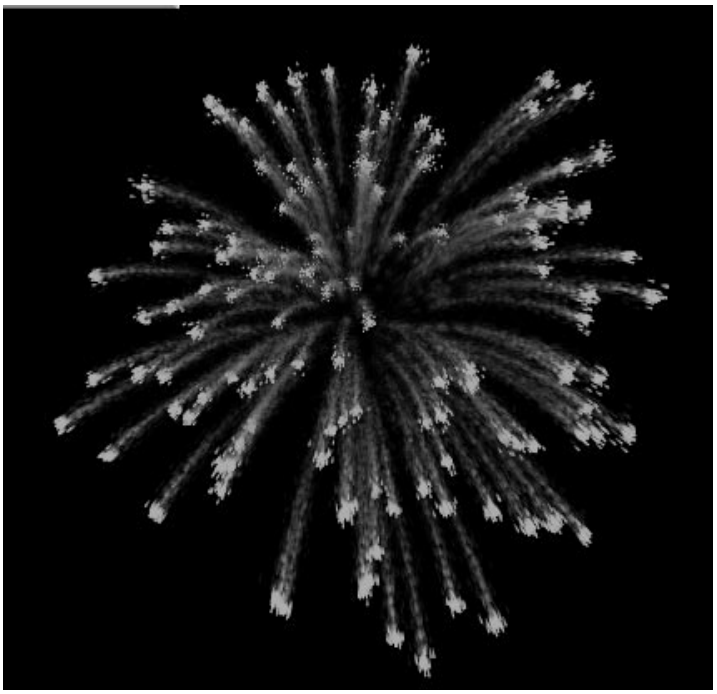
Any other case where you need to guide particles and fields alone do not provide enough control.

9 The Emit Function

Maya provides additional control of particle placement and emission using the MEL (Maya Embedded Language) command called `emit`.

In this lesson you will learn the following :

- Common uses for the emit function
- Common emit syntax and options
- How to use simple conditional statements
- How to add and work with custom attributes
- How to construct strings of commands
- How to use the **eval** MEL command



Chapter 9

Up to this point, the examples have relied on the Particle Tool and the predefined emitters such as surface, directional, omni-direction, etc. to place particles in the scene. For most applications, these provide an adequate starting place.

There are cases where some additional control may be required that is difficult or not possible using the default emitters.

ADDING PARTICLES TO AN EXISTING PARTICLE OBJECT

If you have created a cloud of particles with the Particle Tool and realize you need to add a few more particles to change the shape. This is one common use for the `emit` command.

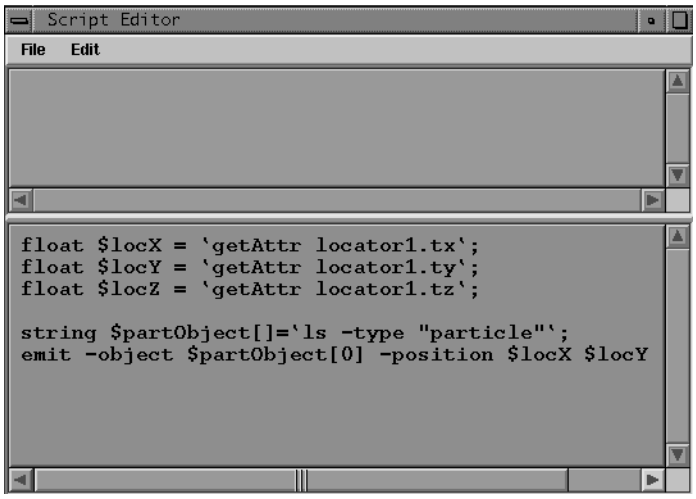
Example 1: Using the emit function with the position flag

1 Create a particle cloud

- Use the Particle Tool to create a random cloud of particles.
- Set the **Render Type** to **Sphere**.
- Set the **Radius** to **0.3.**
- Rename the particles *addParticles.*

2 Add 3 particles to the existing particle object using emit

- Select **Windows** → **General Editors** → **Script Editor**.



The Script Editor

- Enter the following lines in the lower window of the Script Editor:

```
emit -object addParticles -position 1 1 1;
emit -object addParticles -position 2 2 2;
emit -object addParticles -position 3 3 3;
```

- Press the **Enter** key on the numeric keypad :

Each line above adds 1 particle to the existing *addParticles* particle object. The `position` flag is followed by the world space coordinate where the particle gets placed into the scene.

Tip: You could also use the **Command Line** instead of the Script Editor to enter the emit command. The Command Line is the pink text field in the lower left corner of Maya's main window and can be toggled on and off using **Options** → **Command Line**.

Tip: Individual particles cannot be removed from a particle object. However, you can set an individual particle's **opacityPP** to 0 or its **lifespanPP** to 0 using the particleId in an expression or by setting the value in the Component Editor.

Example 2: Use a locator to define particle placement

You can make this process more interactive by setting it up so the `emit` command places the particle at the coordinates of a locator.

- Select **Create** → **Locator**
- Type the following in the Script Editor:

```
float $locX = 'getAttr locator1.tx';
float $locY = 'getAttr locator1.ty';
float $locZ = 'getAttr locator1.tz';
string $partObject[] = 'ls -type "particle"';
emit -object $partObject[0] -position $locX $locY
$locZ;
```

- Use **MMB** to drag the above text to the shelf.
- Move the locator to a location in space where you want to add a new particle
- **Select** the locator and the particle object.
- Click the new shelf button to automatically add a new particle into the selected particle object at the position of the locator.
- Repeat the process as desired.

Tip: Add the contents of the shelf button to a **hot key** to make this even more interactive.

"EMITTING" BASED ON OTHER PARTICLES

The `emit` function can set any attribute for a particle, not just its position. In the following example, `emit` is used to set position and velocity on newly spawned particles to make it appear as though the dying particles are emitting new particles.

1 Create a directional emitter

- Select **Particles** → **Create Emitter**.

  Change **Type** to **directional**
- Rename the emitter *primaryEmit.*
- Rename the particles *primaryParticles.*
- Set the following attributes for *primaryEmit:*

  Spread to **0.25**

  Direction to **0 1 0**

  Speed to **10**

  Rate to **100**
- Set the following attributes for *primaryParticles*

  Render Type to **Sphere**

  Radius to **0.2**

2 Create an empty particle object

- Select **Particle** → **Particle Tool** – ❐, and set the following:

  Number of Particles to **0.**
- Click in any viewport.

Note: If an alert is displayed that no particles are created this can be ignored since your goal here is to create an empty particle object and have the emit command put particles into it later.

- Press the **Enter** key.
- Rename the empty particle object *secondaryParticles***.**

3 Set the Render Type for secondaryParticles

- Select *secondaryParticles* and set the following*:*

  Render Type to **Multi-Streak**

4 Add a lifespanPP to primaryParticles

5 Connect gravity to primaryParticles

- Select *primaryParticles.*
- Select **Fields** → **Create Gravity.**

6 Add a runtime expression to lifespanPP for primaryParticles

- Add the following to lifespanPP's runtime expression :

```
$pos = position;
$vel = velocity;
if ($vel.y<0)
{
lifespanPP = 0;
```

```
emit -object secondaryParticles -position ($pos.x)
($pos.y) ($pos.z) -at velocity -vectorValue ($vel.x)
($vel.y) ($vel.z);
}
```

7 Test the expression

- Rewind; then playback

  Just as *primaryParticles* begin to fall, the `emit` function is invoked and new *secondaryParticles* replace them with the same velocity and position.

Tip: A full description of the flags used by the emit function are listed in the on-line MEL documentation in the scene commands section.

---

8 Edit the expression to add color to secondaryParticles

- Add an **rgbPP** to *secondaryParticles*
- Edit the runtime expression on *secondaryParticles* as shown :

```
$pos = position;
$vel = velocity;
$col = sphrand(1);
if ($vel.y<0)
{
lifespanPP = 0;
emit -object secondaryParticles -position ($pos.x)
($pos.y) ($pos.z) -at velocity -vectorValue ($vel.x)
($vel.y) ($vel.z) -at rgbPP -vectorValue ($col.x)
($col.y) ($col.z);
}
```

9 Test the results

- Press **5** for shaded mode
- Rewind; then playback

  A random color is assigned to each particle in *secondaryParticles*.

CREATING FIREWORKS

You can take the emit function another step further by repeatedly invoking the command in a looping structure to create a firework effect.

1 Create the emitting cannon and the initial particle object

- Create a **Directional** emitter with the following settings :

  Rate to **5**

  Spread to **0.25**

  Direction X to **0**

  Direction Y to **1**

  Direction Z to **0**

  Speed to **10**

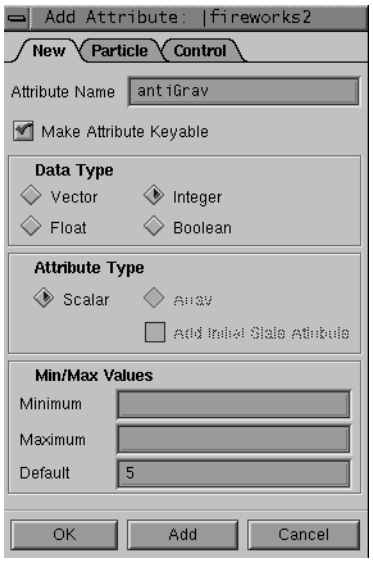
- Rename the emitter *launcher* and the particle object *fireworks1*.

2 Add custom attributes to launcher

Custom attributes are attributes that the user can tailor to his or her specific needs. Below you will add several custom attributes to the emitter that will later be used in particle expressions to modify the motion of the particles and the amount of emission that occurs.

To add custom attributes to the emitter, follow these steps :

- Select *launcher* and choose **Modify** → **Add Attributes**.



The Add Attributes Window

- Add the following custom attributes to *launcher*. All the Attribute Types are **scalar** and **keyable**.

Attribute Name	Data Type	Default Value
antiGrav	Integer	5
showerUpper	Integer	30
showerLower	Integer	30
streamUpper	Integer	1
streamLower	Integer	1

Table of custom attributes to be added to Launcher

3 Add and adjust attributes for fireworks1

- Add a **lifespanPP** attribute to *fireworks1*.
- Add a **per object Color** (rgb) attribute to *fireworks1*.
- Set the attribute values for *fireworks1* as follows :

  Max Count to **3**

  Render Type to **Streak**

  Red to **0.8**

  Green to **0**

  Blue to **0**

  Line Width to **1**

  Tail Fade to **0.1**

  Tail Size to **3.6**

4 Connect fireworks1 to gravity

- Select *fireworks1*.
- Select **Fields** → **Create Gravity.**

Make the secondary particle object

The secondary particle object represents the small projectiles that leave the initial projectile when its velocity reaches 0. These will be created using the emit function.

These secondary particles will act as leading particles for the long streaks of sparks that will be added later. Wherever the leading particles go, the streaks of sparks will follow. The secondary particles are the glowing tips of the streaks.

1 Create the “leading” particle object

- Select **Particles** → **Particle Tool -** ❐, and set the following:

  Number of Particles to **0**

- Click in a perspective or orthogonal view

- Press **Enter.**

  This will create an empty particle object.
- Rename the new empty particle object *fireworks2.*

2 Add/modify attributes for fireworks2

- Add a **lifespanPP**.
- Change the render type to **Sphere** and set a small **Radius** value (**0.05** works well).

3 Create and connect gravity

- Select *fireworks2.*
- Select **Fields** → **Create Gravity**.

Create the final particle object and emitter

Now you'll create the stream of sparks that follow behind the leading particles.

1 Add a directional emitter to fireworks2

- Select *fireworks2***.**
- Add a **Directional** emitter using **Particles** → **Add Emitter.**
- Rename the emitter *sparkEmit* and the new particle object *fireworks3.*

2 Adjust attributes for sparkEmit

- Select *sparkEmit* and set the following attributes :

  Rate to **40**

  Spread to **0.25**

  Speed to **1**

  Direction to **0 1 0**

3 Add a lifespanPP attribute to fireworks3

- In the Attribute Editor *forfireworks3,* create a **lifespanPP.**

  Shortly you will set the value of **lifespanPP** with a creation expression.

4 Set the attributes for fireworks3

- Select *fireworks3* and set the following attributes:

  Render Type to **MultiStreak**

  Line Width to **1**

  Multi Count to **40**

  Multi Radius to **0.3**

  Tail Fade to **.065**

  Tail Size to **3.0**

Add expressions to the particle objects

Now that the particle objects have been built and the appropriate fields connected, you can add expressions to the various particle objects.

1 Use emit to spawn the “leading” particles into fireworks2

- Add the following runtime expression to *fireworksShape1* :

```
vector $pos = fireworksShape1.position;
vector $vel = fireworksShape1.velocity;

float $anitGrav = launcher.antiGrav;
int $upperCount = launcher.showerUpper;
int $lowerCount = launcher.showerLower;
int $upperLife = launcher.streamUpper;
int $lowerLife = launcher.streamLower;

if ($vel.y < 0)
{
fireworksShape1.lifespanPP = 0;
  int $numPars = rand ($lowerCount, $upperCount);
  string $emitCmd = "emit -o fireworksShape2 ";
  for ($i=1; $i<=$numPars; $i++)
  {
    $emitCmd += "-pos " + $pos + " ";
    vector $vrand = sphrand(10);
    $vrand = <<$vrand.x, $vrand.y + $antiGrav,
$vrand.z>>;
    $emitCmd += "-at velocity ";
    $emitCmd += "-vv " + $vrand + " ";
    float $lsrand = rand ($lowerLife, $upperLife);
    $emitCmd += "-at lifespanPP ";
    $emitCmd += "-fv " + $lsrand + " ";
  }

  eval ($emitCmd);
}
```

What does this expression do?

- This expression creates the “flares” at the tips of the fireworks trails.
- For a detailed description of each line in the above expression refer to the **Emit Appendix** at the end of this book.

2 Create a non-dynamic expression to control launcher's rate

A non-dynamic expression is an expression that is not contained within a runtime or creation expression. Non-dynamic expressions are evaluated once per frame.

- Select *launcher* then add the following in the Expression Editor.

```
launcher.speed = rand (12,18);
if (frame%20==0)
launcher.speed = 22;
```

This expression varies the speed at which a particle leaves the cannon so the fireworks explode at different heights. Every 20th frame, a particle gets launched much higher.

Note: The **creation** and **runtime** radio buttons are dimmed in the Expression Editor since the expression is being added to *launcher* which is *not* a particle shape object.

---

3 Assign a random lifespan to the spark trails

To control how long the spark trail burns, add the following to the creation expression of *fireworksShape3*:

```
fireworksShape3.lifespanPP = rand (0.3, 0.7);
```

This causes each particle in the spark trails of the fireworks to die before they are 1 second old.

4 Playback and adjust

After you have played the animation and are comfortable with how this process works, try setting different values for lifespanPP and other particle and attribute values including the custom attributes you added to the emitter.

EXERCISES

- Add **rgb**, **opacity**, and **radius** attributes where appropriate and control them over the particle's age using ramps or expressions.
- Add per point emission so the sparks emit at different rates
- Add a non-dynamic expression that assigns a random number to count so there can be a different number of fireworks in the scene.

CONCLUSION

Now that you've had some exposure to the emit command, you have another tool available for you to achieve the effects you are working on.

The key concepts discussed in this lesson were :

- Using emit to add particles to a particle object
- Using emit to place new particles based on the position of other particles.

- Constructing an emit statement using a looping structure and the eval command.
- Adding an emitter to particles created with emit

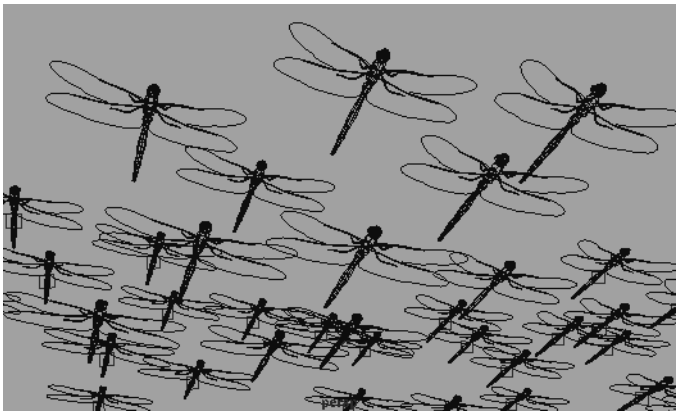
You'll likely come across cases where the methods discussed here are applicable to a situation you are trying to simulate. Be careful about getting side tracked by the more technical approach of using emit if the same effect is easily accomplished using the particle tools available in the interface.

10 Particle Instancing

This lesson focuses on Maya's Particle Instancer which is a tool for substituting particles with geoemtry.

In this lesson you will learn the following :

- How to make geometry match particle movement
- How to add animated geometry to particles
- How to use cycles to instance a sequence of geometry
- How the particle instancer uses custom attributes
- How to add randomness to particle instanced geometry
- Important qualities of Hardware Sprites
- How to create software sprites using the Instancer.



Chapter 10

What is an instance?

An instance in Maya is similar to a duplicated object. The primary difference is that an instance contains no actual surface information but is just a redrawn version of some original object. That original

object acts like a master to all of its instances. The instance takes on all shading and surface characteristics of the original and will update as the original is updated. Since instances contain less information than duplicates, Maya can handle them faster.

What is particle instancing?

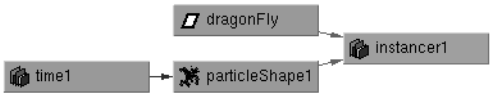
Particle instancing (also commonly called particle replacing) is the process of using particles to control instanced geometry. For example, you could model a honey bee, animate it flapping its wings, then use particle instancing to apply that flapping bee to particles that are swarming through a scene.

Particle instancing is not behavioral animation

Although some complex results can be obtained using particle instancing, it is important not to interpret the particle instancer as a full featured behavioral animation system. For example, you could build a fish swiming then instance that swimming fish onto particles to simulate a school of fish. However, each fish just follows its own particle, there are no behavioral relationships established between the individual instanced elements.

The instancer node

Maya uses the instancer node as the “engine” to perform particle instancing. Although use of the instancer node is not limited only to particles, the most common inputs it receives are from particles and from the geometry that will be instanced to those particles.



Inputs to the Instancer Node in the Dependency Graph

CREATING AN INSTANCER NODE

Example 1: Instancing animated Dragonflies

1 Open the scene file

- Open the file *dragonflyStart.mb.*

2 Animate the wings flapping

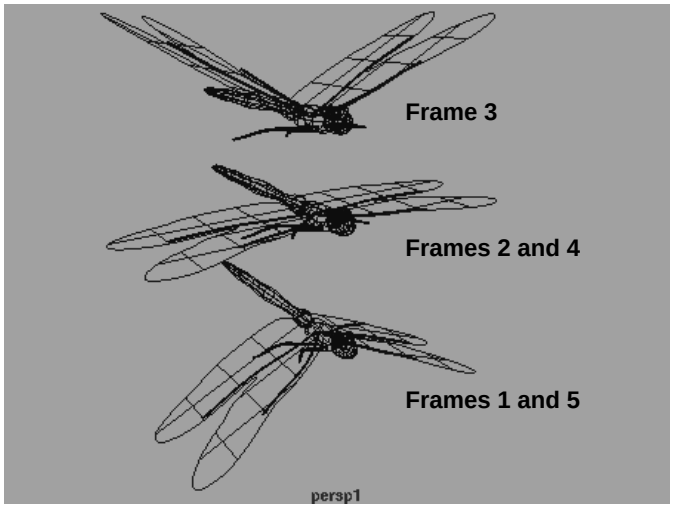
- Select *leftWing* then **shift-select** *rightWing* in the Outliner.
- Use **Shift-E** to keyframe only the rotation channels of both wings simultaneously as follows :

  Frames **1** and **5**, wings are angled down;

  Frames **2** and **4,** wings are straight;

  Frame **3** wings are angled upward;

Tip: You can **MMB** drag in the timeline to advance frames without scrubbing through the animation. This makes it easier to set up one position at two different frame numbers for cycling. You can also use the **RMB** menu in the timeline to cut, copy, and paste keyframes.



dragonFly shown with three different wing positions

3 Create a grid of particles

- Select **Particle** → **Particle Tool** - ❐

  Particle Name to **flyParticles**

  Conserve to **1**

  Number of Particles to **1**

  Create Particle Grid to **ON**

  Particle Spacing to **12**

  Placement to **with textfields**

  Minimum Corner to **-25, 0, -25**

  Maximum Corner to **25, 0, 25**

- Press **Enter.**

  Setting particle spacing to 12 allows enough space in-between each particle so that the instanced dragonflies will not intersect.

4 Randomly offset the flyParticles

- Add the following Creation Expression to the position attribute of **flyParticles**

```
float $randY = rand (-3,3);
float $randXZ = rand (-1,1);
vector $offset = <<$randXZ, $randY, $randXZ>>;
```

```
position = position + $offset;
```

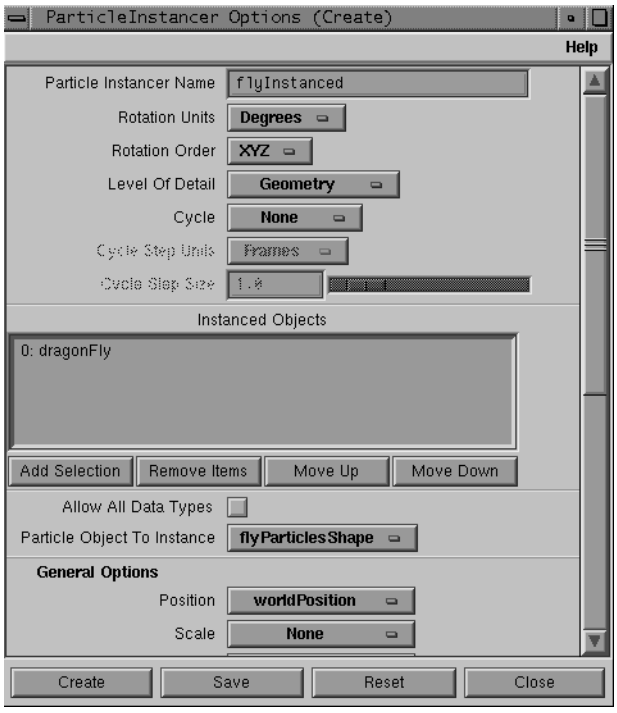
- Press **Create** to offset the particles from the grid. You can repeatedly offset the particles by pressing the rewind button.
- With the particles selected, set initial state by selecting **Solvers** → **Initial State** → **Set for Selected**.
- **MMB-drag** the contents of the expression to a shelf then delete the expression and close the Expression Editor.

5 Instance the dragonfly to the particles

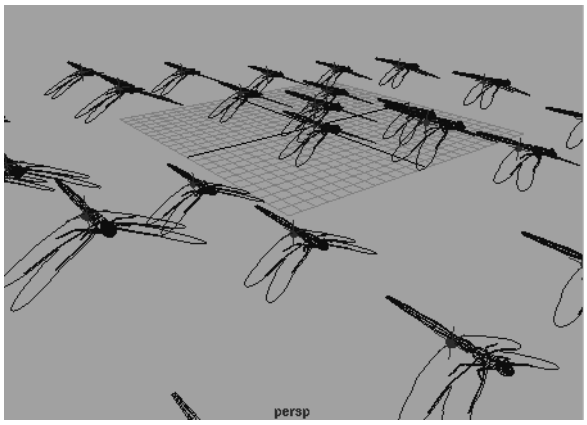
- In the Ouliner, select **dragonFly**
- Select **Particles**→**Particle Instancer** - ❐, and set the following:

  Particle Instancer Name to **flyInstanced**
- Press **Create**

  This creates an Instancer node in the scene and creates an instanced version of the dragonfly for each particle.



The Particle Instancer option window



The instanced dragonflies

6 Hide the original dragonfly object

- Select *dragonFly*.
- Select **Display** → **Hide** → **Hide Selection.**

7 Add a vector attribute to the particles

Now you will add some variation to the size of each of the instanced dragonFlies.

- Open the Attribute Editor for **flyParticlesShape.**
- Press **General** in the Add Dynamic Attribute section, and set the following:

  Attribute Name to **bugScaler**

  Data Type to **Vector**

  Attribute Type to **Array**

  Add Initial State Attribute to **On**
- Press **Ok**.

8 Assign values to bugScaler with a creation expression

- **RMB** in the **bugScaler** field and select **Creation Expression**
- Add the following to the Expression Editor:

```
if (frame==1)
{
$rand = rand (0.4, 1.5);
bugScaler = <<$rand, $rand, $rand>>;
}
```

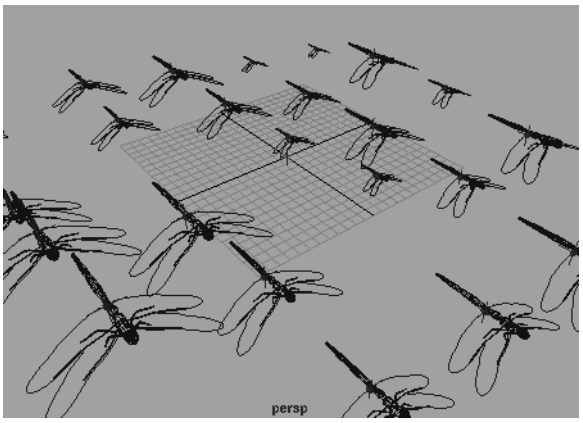
  This expression picks a random number between 0.4 and 1.5 for each dragonfly then assigns that value to the bugScaler attribute.

9 Set the instancer to use bugScaler to scale each instance

- Open the Particle Instancer section of the Attribute Editor.
- Under General Options, set **Scale** to **bugScaler**.

Any attribute added to the particle object can be fed into the various control attributes of the instancer node. In this case, you are computing a value with an expression, storing that value in the bugScale attribute, then assigning bugScale to the scale option in the instancer node. The ability to use any attribute value for any of the connections to the instancer is what gives the the instancer its flexible control since you are able to control the contents of those attfibutes with your own expressions.

Note: Since bugScale is set at frame 1, the scale of the bugs will change everytime the rewind button is pressed. Once you are satisfied with the scale of the bugs, you can delete or comment out the expression. Use two forward slashes (/ /) at the beginning of each line to comment out the expression.



Instanced dragonflies with random positioning and scaling

10 Set initial state (Optional)

If the bugScale attribute alwasy resets to 0 upon rewind (the bugs disappear). Select the particle and set initial state. It is a good idea to save your file before setting initial state.

- Select the particles.
- Select **Solvers** → **Initial State** → **Set for Current.**

11 Apply a uniform field to the particles then playblast

- Use the following settings for the uniform field

  DirectionX, **Y** and **Z** to **1, 0, 0** respectively

  Magnitude to **15**

  Attenuation to **0**

- Use **Windows** → **Playblast -** ❐ to preview the motion of the dragonflies.

Tip: If you wanted each particle to move differently, you could apply an acceleration runtime expression to the particles instead of using a uniform field.

Example 2: Cycling through a sequence of butterflies

1 Open the scene

- Open the file *butterfly.ma.*

  This scene contains one butterfly with no animation on it. You will create an animated cycle using the instancer and several duplicates of the butterfly.

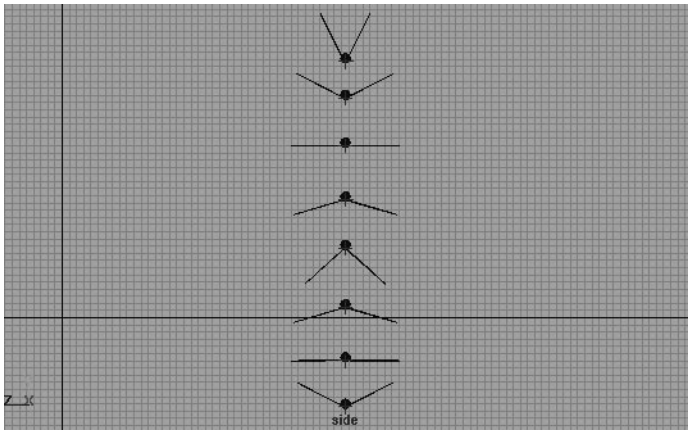
2 Setup a cycle using duplicated snapshots of the butterfly

- Use the Outliner to select *origButterfly*.
- Make 8 duplicates of the object.
- Hide *origButterfly*.
- Position the wings of each duplicate as shown in the image below.

  To rotate both wings simulataneously, select *polyRightWing*, **Shift- select** *polyLeftWing* then use the **Rotate** tool.

Note: Do not drag-select a box around the butterfly and move it. This causes the items below the top hierarchy to be offset from the base object's coordinate system and will cause offseting from the particles during instancing. Instead, select the top node of the duplicated object in the Outliner before moving it.

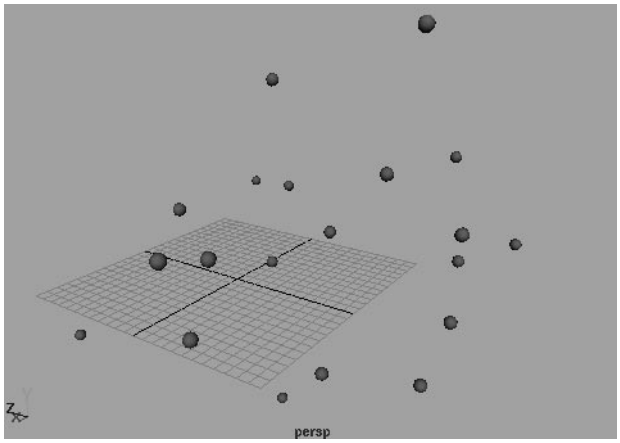
- Name the duplicated butterflies *postion1*, through *position8.*



8 butterflies producing one complete flapping cycle.

3 Create particles in the scene

- Use the Particle Tool to create a cloud of about 20 particles with a maximum radius around 20.
- Rename the particles *butterflyParticles*.



Cloud of 20 particles to be used for butterfly instancing

4 Set basic instancer options

- Use the Outliner to select *Position1* through *Position8*. Make sure *Position1* is selected first and *Position8* is selected last since the order they are selected will determine the cycling order used by the Instancer node.
- Select **Particles** → **Particle Instancer** - ❒, and set the following options :

  Particle Instancer Name to **butterflyInstancer**

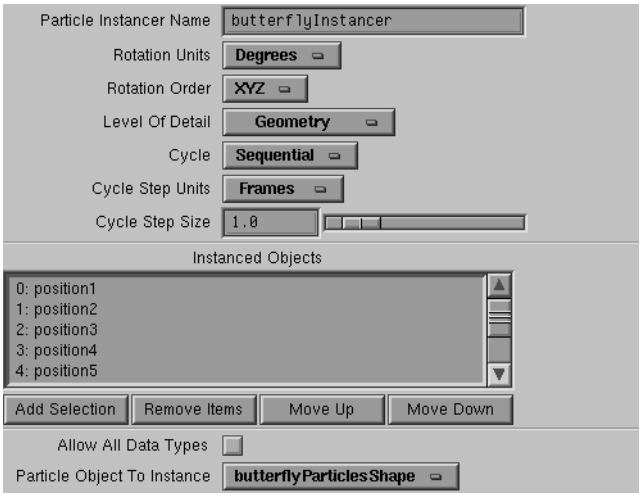
  Cycle to **Sequential**

  Cycle Step Size to **1**

- Press **Create**.

  The list of objects the Instancer will use is shown in the Instanced Objects list. The number beside the object is called the **object index**. The first object in the list is always index 0. The Instancer uses this index value to determine which object in the sequence of butterflies to display at a given point in time.

  A cycle setting of **Sequential** causes the Instancer to cycle through the object indicies in sequence rather than not using any cycling at all. A **Cycle Step Size** of **1** causes the Instancer display each object index for 1 frame before changing to the next item in the list.

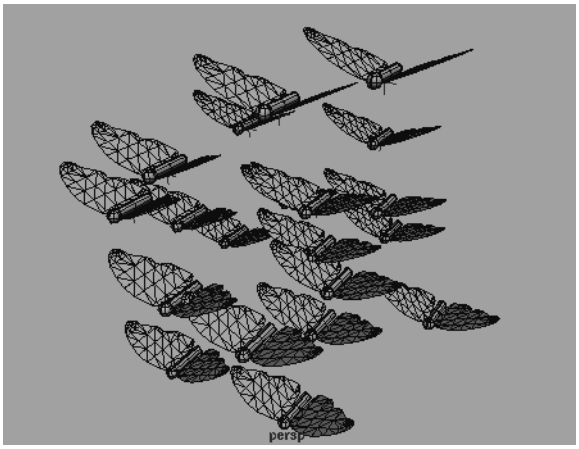


List of instanced objects and their object indicies in the instancer window

5 Make a test run

- Select and **Hide** *position1* through *position8*
- Playback to view the instanced objects cycling

  All butterflies cycle through the 8 positions in exactly the same fashion.



Instanced butterflies all with the same initial object index

6 Add a custom attribute to control cycling on a per particle basis

It was previously stated that the **Cycle Step Size** determines how long the instancer will display each object index before switching to the next item in the list. Since the particleId is unique for each particle, it can be used to control this duration on a per particle basis. This is accomplised by muliplying the particleId by the age and storing the result in a custom attribute which, in turn, gets fed into the Age control of the Instancer's cycling options as explained below.

- Select *butterflyParticles.*

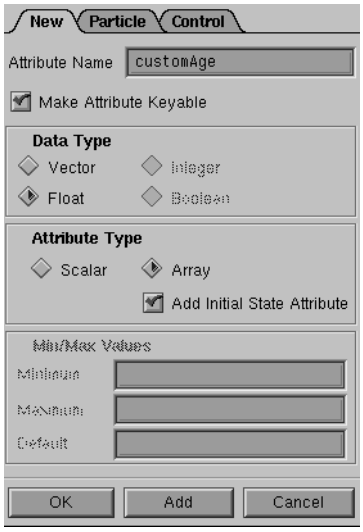
- Press the **General** Button in Add Dynamic Attribute section.
- Set the following options:

  Attribute Name to **customAge**

  Data Type to **Float**

  Attribute Type to **Array**

  Add Initial State Attribute to **ON**
- Press **OK.**



Adding customAge as a float array attribute

7 Assign values to customAge with a runtime expression

Now that you've added the attribute, you need to assign values to it for each particle. Since age changes over time, you'll use a runtime expression instead of a creation expression

- **RMB** on the *customAge* field and select **runtime expression.**
- Enter the following runtime expression in the Expression Editor:

```
if (particleId == 0)
   customAge = age;

else if (particleId == 1)
    customAge = age * 0.5;

else if ((particleId % 2 == 0) && (particleId % 3 == 0))
   customAge = age * 0.25 * particleId / 4;

else if (particleId % 2 == 0)
```

```
   customAge = age * 0.4 * particleId / 4;

else if (particleId % 3 == 0)
   customAge = age * 0.35 *particleId / 4;

else
   customAge = age * 0.2 * (particleId) / 4;
```

This expression assigns a different value to *customAge* based on multiplication of the particleId. Since these particles are not being emitted they have the same Age. To get around this, Age is multiplied by a decimal value which was arbitrarily selected to provide variation in Age for each particle. This value is then multiplied by the particleId which is, again, another unique value. The particleId is divided by 4 to keep the values small. If this division wasn't done, the values in customAge would get large quickly and cause the cycle to occur too fast.

8 Set particle render type to numeric

To get a better idea of which portion of the expression is controlling which particles, you can use the numeric render type.

- Select *butterflyParticles.*
- Set **Render Type** to **Numeric**

  By default, the numeric Render Type displays the particleId attribute for each particle. This makes it easier for you to see which particle will be affected by which portion of the expression. For example particleId 12 is evenly divisble by both 3 and 2 so it will therefore be set by this portion of the expression :

  ```
  else if ((particleId % 2 == 0) && (particleId % 3 == 0))
     customAge = age * 0.25 * particleId / 4;
  ```

  for the evaluation of customAge.

Tip: You can change which attribute the numeric render type displays by pressing the Current Render Type button in the Attribute Editor then typing in the new attribute name in the field provided.Attributes such as rgbPP and mass are other useful attributes to view in numeric mode.

---

9 Set the Cycle Options, Age attribute to customAge

- In the Instancer section of butterflyPartcles, choose **customAge** from the **Cycle Options** → **Age** pulldown menu.

10 Add a float array attribute to control the starting object index

Currently, all butterflies begin their sequence from object index 0 (position1.) They all cycle through the instanced object list starting from position1 and ending at position8 then repeating.

You can change the object index the sequence starts on by setting the **CycleStartObject** attribute in the Instancer. In this case, you will create another custom attribute called *startPick* to control **CycleStartObject** on a per particle basis similar to the way you set up *customAge*.

- Add a custom float array attribute to *butterflyParticles* called *startPick*
- Add the following **creation expression** to *startPick*. The value chosen for *startPick* here will determine which butterfly the instancer chooses to begin the cycle on.

```
if (particleId == 0)
   {
      startPick = 0;
   }

else if (particleId == 1)
   {
      startPick = 1;
   }

else if ((particleId % 2 == 0) && (particleId % 3 ==
0))
   {
      startPick = 2;
   }

else if (particleId % 2 == 0)
   {
      startPick = 3;
   }

else if (particleId % 3 == 0)
   {
   startPick = 4;
   }

else
   {
      startPick = 5;
   }
```

11 Set the Cycle Options, cycleStartObject attribute to startPick

- In the Instancer section of butterflyPartcles, choose **startPick** from the **Cycle Options** → **cycleStartObject** pulldown menu.

12 Add uniform and radial fields to move the particles

You can add fields or expressions to control the motion of the individual particles as desired.

Planning, optimizing, and rendering considerations

Now you have used the Instancer with an animated object and a cycled sequence of "snapshots". There are advantages and disadvantages to both methods. Using an animated object allows you to use 2D and 3D motion blur when rendering. Motion blur is not available when cycling through a sequence of instanced objects. Therefore, it is best to consider rendering requirements when setting up shots requiring instancing.

It is also important to keep your geometry as simple as possible. Making your surfaces single sided and keeping NURBS patches low (or poly count low) makes a big difference.

One advantage to using a sequence of "snapshots" is that you have control over the duration of each snapshot and also the starting point of the cycle.

Hardware Sprite Render Type

The Sprite Render Type is used for displaying 2-D file texture images on particles.

- The scene file *snowSpriteHW.mb* illustrates a simple application of hardware sprites. Open this file and playback the animation.

  A phongE shading group with a file texture of a snowflake with an alpha channel is assigned to the particles.

  If you tumble the camera, the sprite images will always aim at the camera. This is a built-in feature of hardware sprites.

Creating Software Sprites with the Instancer

The hardware sprite render type is only available for rendering using the Hardware Render Buffer.Therefore, you cannot render 3-D motion blur, reflections, refractions, or shadows like you can can when using software rendering. Here, we'll discuss a method for creating software renderable particle "sprites" using the Instancer node. This will allow you to take advantage of these important software rendering features while maintaining the core functionality that the hardware sprite render type provides.

Setup: Worldspace up vector camera setup

Since sprites should always face directly at the camera, you will need to set up your camera so there is some information available to tell the sprites what to point at. You will eventually use a polygon plane, particles and the Instancer node to make the software "sprite" objects.

1 Create a new scenefile

2 Create a camera at the origin

- Select **Primitives** → **Create Camera.**

For this example you will use a single node camera.

- Rename the new camera *spriteCam*.

3 Create a locator at the origin and make it a child of spriteCam

- Select **Primitives → Locator.**
- Rename this locator *camLocal.*
- Parent the locator under *camLocal.*

  camLocal represents the local coordinate system of *spriteCam* at the back of the lens of the camera.

4 Duplicate the camLocal locator

- Duplicate the *camLocal* locator.
- Rename the new locator *camUpLocal.*
- Move it up one or two units so that it is directly above *camLocal.*

  camUpLocal will be used to find out which direction is up for spriteCam within its own local coordinate system.

5 Create another locator at the origin

- Create another locator at the origin
- Rename this locator *camWorld.*
- This locator should *not* be placed within the *spriteCam* hierarchy

  camWorld will be used to find out what world space coordinate the center of the camera lens is at.

6 Duplicate the camWorld locator

- Duplicate the camWorld locator.
- Rename this locator *camUpWorld.*
- *camUpWolrd* will be used to find out what world space coordinate is considered up for the camera.

7 Point Constrain camWorld to camLocal

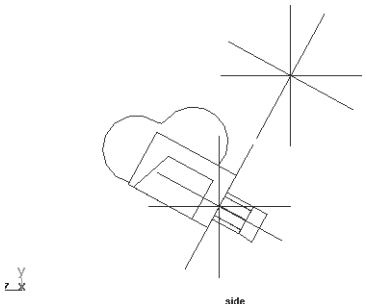
Point constraining *camWorld* to *camLocal* will cause *camWorld* to always follow the position but not the rotational orientation of *camLocal*.

- **Shift-select** *camLocal* then *camWorld.*
- Select **Constrain → Point Constraint.**

8 Point Constrain camUpWorld to camUpLocal

Point constraining *camUpWorld* to *camUpLocal* will cause *camUpWorld* to always follow the position but not the rotational orientation of camUpLocal.

- **Shift select** *camUpLocal* then *camUpWorld***.**
- **Constrain → Point Constraint.**



Side View of spriteCam With Locators for World and Local Axes

Instancing Setup

Now you will add custom attributes to the particle object. These custom attributes will later be selected as options in the pop-up menus of the Instancer.

1 Add vector array attributes to particleShape1

- Press **General** under Add Dynamic Attributes.
- Add 4 **Vector Array** attributes named as follows :

  spriteWorldUp

  spriteAimPos

  spriteAimAxis

  spriteAimUpAxis

2 Add a creation expression to particleShape1

- **RMB** select **Creation Expression** in the *spriteWorldUp* attribute field.
- Enter the following expressions :

```
spriteAimAxis =  << 0, 1, 0 >>;
spriteAimUpAxis =  << 0, 0, 1 >>;
vector $camPos =
<<camWorld.tx,camWorld.ty,camWorld.tz>>;
vector $camUp =
<<camUpWorld.tx,camUpWorld.ty,camUpWorld.tz>>;
vector $upDir = $camUp - $camPos;
spriteAimWorldUp= $upDir;
```

3 Add a runtime expression to particleShape added attributes

- Enter the following as a runtime expression on the particleShape node:

```
spriteAimPos = <<spriteCam.tx, spriteCam.ty,
spriteCam.tz >>;
vector $camPos =
<<camWorld.tx,camWorld.ty,camWorld.tz>>;
```

```
vector $camUp =
<<camUpWorld.tx,camUpWorld.ty,camUpWorld.tz>>;
vector $upDir = $camUp - $camPos;
spriteAimWorldUp = $upDir;
```

4 Create Particle Instancer for the plane object

- Select the *spritePlane* object.
- Select **Particle** → **Particle Instancer.**

5 Set Instancer options

You have put all the pieces in place. Now hook the particle attributes up to the instancer.

- In the *particleShape1* Attribute Editor open the Particle Instancer section, and set the following options:

  General Options :

  Position to **WorldPosition;**

  Rotation Options :

  AimPosition to **spriteAimPos**

  AimAxis to **spriteAimAxis**

  AimUpAxis to **spriteLookUp**

  AimWorldUp to **spriteAimWorldUp**

Note: If NONE is selected for AimDirection a default value of <<1,0,0>> will be used. If NONE is selected for AimPosition <<0,0,0>> is used. If NONE is selected for AimAxis <<1,0,0>> is used. If NONE is selected for AimUpAxis <<0,1,0>> is used.

---

6 Test the particle instancing

- A good test is to turn your **Render Type** to **Sprite**.

  You should see that the instanced plane matches perfectly.

How do these expressions work?

- Determining the `spriteAimWorldUp` vector:

```
vector $camPos =
<<camWorld.tx,camWorld.ty,camWorld.tz>>;
vector $camUp =
<<camUpWorld.tx,camUpWorld.ty,camUpWorld.tz>>;
```

- `$camPos` is determined from the current camera position in world space. The expression gets this value from the point constrained locator `camWorld`. This is a vector value.
- `$camUp` is determined from the current position of the `camUpWorld` locator that is offset from the `camWorld` locator.

```
vector $upDir = $camUp - $camPos;
spriteAimWorldUp = $upDir;
```

- `$upDir` is determined from the difference between camWorld and camUpWorld. This is a vector value.
- `spriteAimWorldUp` is given this difference.

As the camera moves and rotates in world space, `spriteAimWorldUp` is reduced to a single vector that is used to compute each instanced objects current reference point for what is straight up. Since the camera may have rotated, what is straight up to the camera is not the same as what is straight up to the rest of the Maya world. Without this information the instancer would just assume that world up is the same as the Y up presented by the Maya world up setting and the sprites would orient themselves to the wrong up reference point.

7 Ready for Rendering

- Now that you have established the control of the software sprites, they are ready for software rendering you can apply shaders and render with shadows, raytracing, motion blur, etc.

Excersies

- Combine the butterflies and the dragonflies together into one scene using two seperate instancer nodes.
- Create a custom attribute called customScale to control a random proportional scale of each butterfly.
- Write an acceleration expression to move the butterflies through your scene instead of using fields.
- Create a cloud of particles and give them random swarming motion using an acceleration rule. Instance these swarming particles to another group of moving particles to make several collections of swarming particles.
- Make the butterflies randomly flutter then glide for a few frames

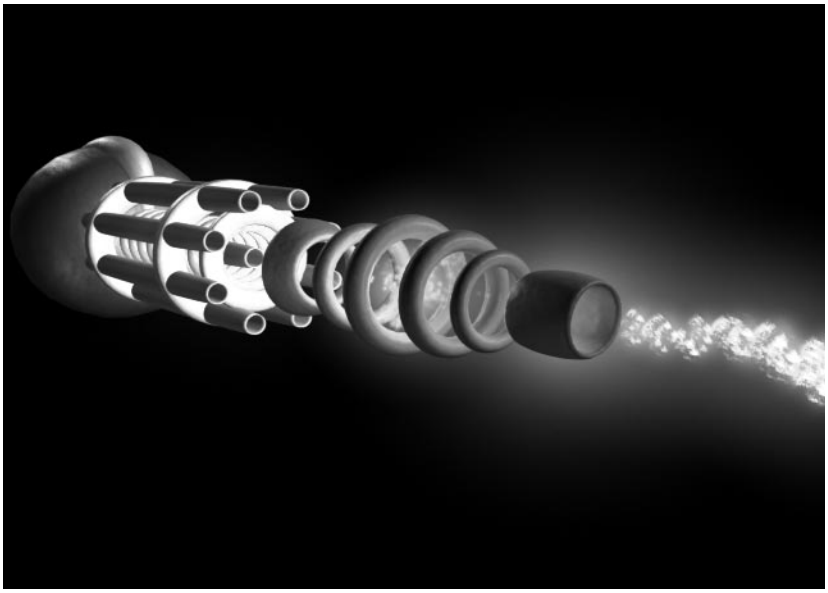
Questions to test your understanding :

- What is Particle Instancing?
- What is an object index?
- What happens if you increase the Cycle Step Size?
- What does the Allow All Data Types option do?
- Explain why butterflies instanced to a higher particleId flap their wings faster then those instanced to a lower particleId? How can you speed up the flapping of the butterflies instanced to lower particleIds?

11 GOALS

This lesson focuses on working with Maya's particle goal functionality. In this lesson you will learn the following:

- Creating Goal Objects
- Goal Parameters
- Fun with Goals
- Per Particle Goal Attributes and functionality
- Particle Hair



Chapter 11

PARTICLE GOALS

One of the more powerful methods that you have for controlling particle position and motion is through the use of Goals. You can create goals out of curves and geometry. A particle can also have multiple goal objects.

When a Goal is created, attributes are added to the *particleShape*. These attributes control the amount of weight of a goal objects influence and the smoothness of the goal/particle interaction.

Particle goals are a big part of soft body dynamics. You will be looking at goals for soft bodies in the soft body section as well. The concepts covered in this lesson will be directly employed in the soft body lessons.

Creating Particle and Non-Particle Goals

Creating a Particle Goal Object involves selecting the particle then the object or objects that will be used as the goal objects and then selecting from the menu **Particles** → **Add Goal**. You have the option of using particles or geometry objects as the goal objects. You can also select the CVs of the geometry as the goal or the transform of the object/s as the points in space that the particles will move towards. When more than one object will act as goals for a particle the resulting goal will be a combination of the goal objects position and the weights that have been set for each goal on the particleShape.

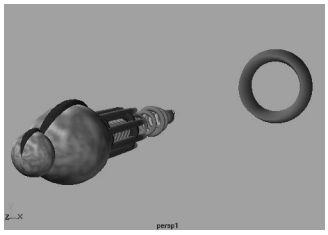
Goal weights and goalPP

Goal weights can be determined from a particle object standpoint or on a per-particle basis. The per-particle goal weight is determined from the value of the goalPP attribute. This is a dynamically added attribute. The goalPP weight is then added to the goal weight of the particle object for a total particle goal weight.

Goal Smoothness

Goal Smoothness controls how particles accelerate toward a goal object. A goal smoothness of 0 will produce a linear or constant acceleration while a goal smoothness of 10 will produce a ease in and ease out type of acceleration.

Example: RayGun



In this example you will control particle movement with goal objects only. You will notice that many advantages can be gained from using goals in place of expressions and fields.

1 Open the scene file

- Open the file *rayGun_geometry.mb*

This scene file consists of a *rayGun_group* and a target wall. The *rayGun_group* has several objects underneath as children:

Gun - This is the group that holds the rayGun geometry.

targetFocus - This is the goal for the particles.

circleEmitter - This is the particle emitter.

pointLight - This is the gun's light source.

coneControl - This is another object that is used for particle control.

2 Add a curve emitter to the curveEmitter object

This emitter will serve as our particle source.

- Select the *curveEmitter* object.
- Select **Particles** → **Add Emitter-** ❐, and set the following:

  Emitter type to **Curve**.
- Press **Add.**

3 Add the targetFocus object as a goal to the emitted particles

This object will be the main destination for the emitted particles. Notice that it has a **Post Infinity Linear Rotation** applied.

- In the Outliner, **Cntrl-select** the particle object then the *targetFocus* object.
- Select **Particles** → **Add Goal.**

4 Playback to see the results

- View *particleShape1* Goal attributes in the attribute editor.
- Note attributes for **Goal Weights and Objects**:

  Goal Smoothness

  targetFocusShape

  Goal Active
- Experiment with these values.

  Attributes that affect particle motion in general and their affect on particles being moved by goals are listed below:

  Dynamics Weight

  Conserve

  Level Of Detail

  Inherit Factor

  Play with these values and combinations of these values in concert with **Goal Smoothness** and the *targetFocusShape* **weight**.

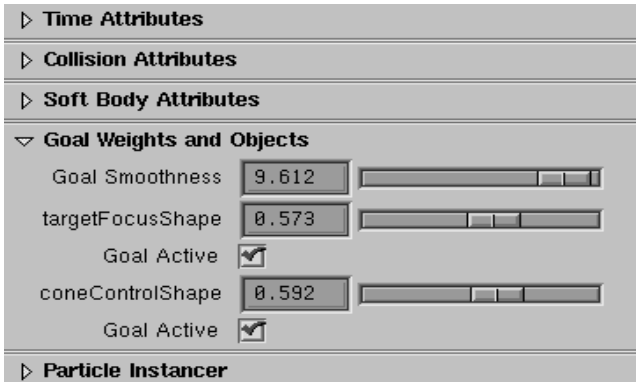
  Conserve is a very important attribute for controlling the acceleration of particles towards their goals. If you find that particles are overshooting there goal object you may be able to dampen there movement with this attribute.

  Dynamics Weight may not have much of an effect but notice that if it is zero the simulation will not compute.

Inherit Factor controls the particle's coupling to initial state and emission speeds.

Tear Off a copy of the Emitter's attribute editor and adjust emission attributes for the curve emitter while you are adjusting the particleShape attributes.

The main attributes at work are the **Conserve, Goal Weight** and **Goal Smoothness**. Experiment with these attributes to get a feel for how the particles are being driven.



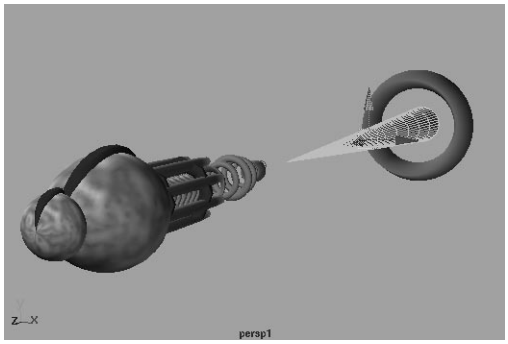
Goal Attributes on the ParticleShape

- Repeat this process for the other booms

5 Add coneControl as another particle goal object

- Add *coneControl* object as another goal object for the rayGun particles.

  Experiment with adding in this objects goal weight to the particle object.



Tip: To remove a goal objects influence, toggle off Goal Active.

6 Experiment with different Animation, Scale And Position settings on the Goal objects.

7 Note a goalPP attribute was added to the particleShape

Maya adds the goalPP attribute to the particleShape when a goal is created for the particle. This attribute allows you to set goal weights on a per-particle basis. The total goal weight is the object Goal Weight added to the goalPP value. By default it is created with a value of 0.

8 Animate the rayGun_group

- Animate the *rayGun_group* as if the gun is tracking an object or trying to disintegrate a moving target.
- Set keyframes at frames 1 and 100. You may want to use Gimbal rotate manip to move the rayGun as if it were on a turret.

9 Parent the particle object into the rayGun group

When you animate the *rayGun_group* transform all of the child objects including the goal objects will translate and rotate together. The particles will move towards their respective goals but will react in world space.

- Drag and drop the particle object into the *rayGun_group* in the Outliner.

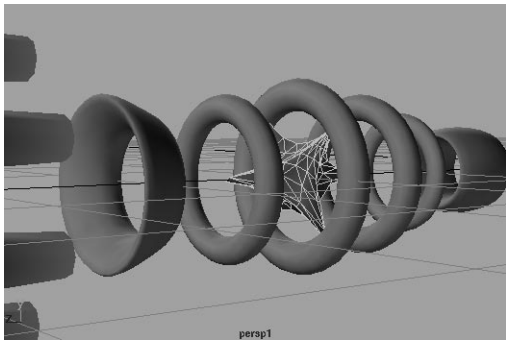
Optional Exercise: Thrust chamber particle effect

The particles need to be concentrated before they can be fired. The thrust coil is the source for the particle energy but the concentrating rings are where the particle reaction occurs. By using a polygon as a goal target you get a more ordered dispersion to the poly vertexes. This avoids the particles trying to go to the CVs on a NURBs object.

Hints:

Create another circle emitter to act as the thrust source.

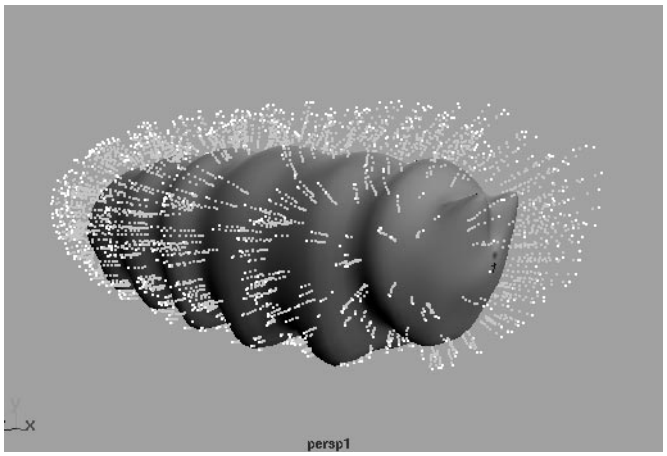
Use the stream circle emitter as another goal to pull on the particles beyond the star target.



Example: Chia Caterpillar

In this exercise you will grow a furry Caterpillar's hair. The idea here is to use a surface as an emission source and another surface as a goal object for

the particles to move towards. Both surfaces can be bound to a skeleton. We will use the per-particle attributes goalU, goalV to control where on the outer surface the particles will grow to. parentU and parentV per-particle attributes keep track of where a particle was emitted from. By passing this information into the goalU and goalV attributes we can direct the emitted particle directly to the goal object on the same UV position.



Catapult with spring and hinge constraint

1 Open scene file

- Open the scene file *caterpillar.mb*

  This file contains two objects:

  caterpillar

  caterpillarGoal

2 Add surface emission to the caterpillar object with ParentUV on

- Select *caterpillar* from the outliner.
- Select **Particles** → **Add Emitter** - ❐, and set the following:

  Emitter Type to **Surface**
- Press **Add.**
- In the attribute editor select **Need Parent UV**

  This will enable the parent UV information to be attached to the emitted particles. The UV emission point information is stored and can be accessed using the parentU and parentV attributes on the particle object.

3 Add caterpillarGoal as a goal object for the emitted particles

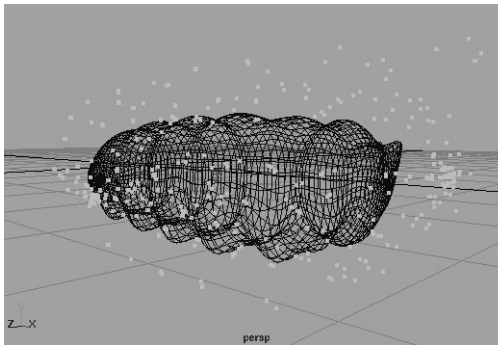
- **Cntrl-Select** in the Outliner, the particle object then the *caterpillarGoal* object
- Select **Particle** → **Add Goal.**

  This will make *caterpillarGoal* a goal for the emitted particles.

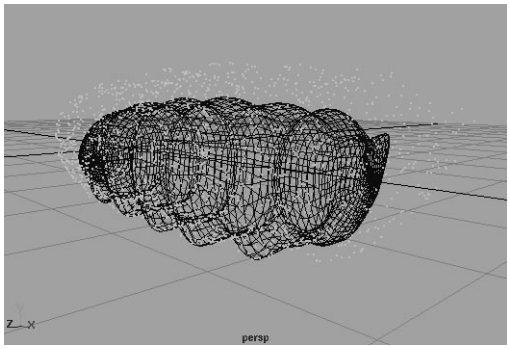
4 Add per-particle goal attributes to the particleShape

To further control the behavior of the particles as they travel to their goal object we can add per-particle attributes to the particle object. **goalU** and **goalV** and **parentU** and **parentV** can be added in addition to the **goalPP** attribute that is created automatically when a goal is added.

- Select the *particle1* object**.**
- In the Attribute Editor under the Add Dynamic Attributes section:
- Press **General** button.
- In the Add Attribute window, click on the Particle tab.
- Select **goalU** from list.
- Press the **Add** button.
- Repeat this for the attributes **goalV, parentU, parentV** and **parentId.**



Particles going to the surface CVs



Particles going to the goalU and goalV via parentU and parentV

5 Create creation expressions to control the goal surface location

In the Per-Particle Array section of the attribute editor for *particle1* create a creation expression for **goalU** and **goalV** attributes.

- Enter the following into the Expression Window:

```
particleShape1.goalV = particleShape1.parentV;
particleShape1.goalU = particleShape1.parentU;
```

These expressions set the particle goal UV targets to be the same as the parent UV positions on the surface. The particles will travel towards the surface's random UV coordinates instead of the CV's which is the default goal position.

6 Add a directional emitter to particle1

You are going to use the first *particle1* object to emit a trail of particles as it moves out to its goal position.

- Select *particle1*.
- Select **Particles** → **Add Emitter -** ❐, and set the **Emitter Type** to **Directional**.

7 Add goals caterpillar and caterpillarGoal to particleShape2

- In the Outliner, Select *particleShape2* then **cntrl-select** *caterpillar* and *caterpillarGoal* objects.
- Select **Particles** → **Add Goal.**

8 Add goal attributes to particleShape2

You will want the emitted particles to move towards the same spot that the emitting particle came from. To do this we will need to keep track of which particle emitted the *particleShape2* particle and what the goalU and goalV of the emitting particle are. The parentId attribute is responsible for keeping track of who the parent is and you will use this attribute to access the parent particles attributes of goalU and goalV.

- Add these attributes to *particleShape2*:

  goalU

  goalV

  parentId

9 Add a creation expression for goalU and goalV of particleShape2

You may find several parts of this expression interesting. Firstly, since it is an expression, there are a few hoops you will need to jump through to build the variable that will feed the parent particle's goal information through to the child particle.

```
int $parentId = particleShape2.parentId;
float $goalU[] = ‘particle -id $parentId -at goalU -q
particleShape1‘;
particleShape2.goalU = $goalU[0];
float $goalV[] = ‘particle -id $parentId -at goalV -q
particleShape1‘;
particleShape2.goalV = $goalV[0];
```

The first variable, `$parentId` is declared as the `particleShape2.parentId`. This is necessary because we cannot use just `parentId` as an argument to the -id flag.

The second variable is a float array that gets the result of the particle command used to query the particleShape1 particle's goalU value based on the parentId value.

This value is placed into the first spot of the float array, [0]. Which is then passed into the `particleShape2.goalU` attribute.

This is repeated for the goalV value as well.

10 Animate the caterpillar

The caterpillar has been setup for animation with joints and IKSpline. The clusters provide handles to manipulate the caterpillar joints and thus the caterpillar.

SUMMARY

- Creating goal objects
- Setting goal weight and smoothness
- Goals and particle hierarchy
- Working with Per Particle goal attributes
- goalU and goalV attributes
- parentU and parentV attributes

Goals are an intuitive and fun method of particle manipulation. These can also solve particle movement problems that only very intense expressions can mimic.

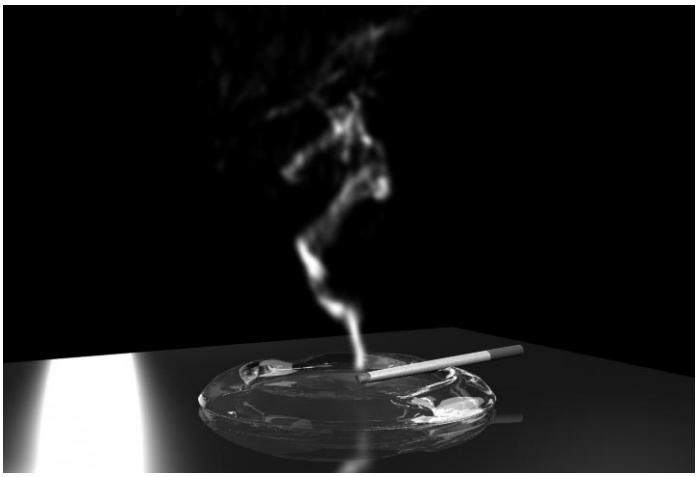
Chapter 12

12 Rendering Dynamics

This lesson covers rendering techniques for particles and dynamic objects.

In this lesson you will learn the following:

- Hardware Particle Rendering Types
- Software Particle Rendering Types
- Particle Caching
- Particle Render Utilities
- Particle Age Mapper
- Particle Cloud Rendering



Particle Render types

Maya provides two types of rendering for particles. Hardware and Software.

Hardware rendering uses the graphics buffer and graphics memory of your computer to draw the image to the display and then take a snapshot of this image. This snap shot is then written to a file as a rendered image. This technique of using the hardware rendering capabilities of your computer has the advantage of being very fast but also the limitation of few rendering perks like shadows, reflections and post process effects like glow. Often particles are rendered only for the positional and matte or alpha information using the hardware renderer. The the actual look of the particle effect is obtained by adding color, shadows, reflections or environment lighting in the compositing stage of production. Again speed and flexibility for asthetic change of mind is behind this pipeline of image creation.

The hardware render types in Maya are the **point, streak, sphere, sprite** and **numeric** render types. As a subset of these types there are also two versions of point and streak that utilize multi-pass rendering. These are the **multi-point** and **multi-streak** particle render types.

The software render types in Maya are the **blobby surface, cloud** and **tube** render types. These render types allow for various combinations of surface and volumetric shading techniques. Their shaders are constructed from the same shading nodes that are applied to geometry and lights.

When you software render, any hardware render type particles are skipped. When you hardware render, software render type particles are rendered as there respective hardware display appearance, filled circles.

HARDWARE RENDERING

The Hardware Render window can be found under **Windows** → **Rendering** → **Hardware Render Buffer...** This window is a free floating window that will assume the size of the selected resolution format. It is adviseable to make sure there are no windows beneath or in front of this window when you are rendering to it. It is also advisable to use an absolute black desktop background if you are going to be doing alot of hardware rendering.

Hardware rendering can also be used as a quick animation test. Geometry can be hardware rendered with lighting and textures but without shadows or advanced lighting effects. There are also many options that allow geometry matting to be generated to aid in the compositing process

Multipass Hardware Rendering

Multipass hardware rendering creates a softer rendered look for your particles. It can also anti-alias your geometry that is being hardware rendered. Multipass rendering requires you to use multipass render type as your particle render type. You have the choice of multi-point or mult-streak.

For each of these render types you will have several particle attributes that control the multi pass effect:

Multi Count: Controls the number of added and offset psuedo particles

Multi Radius: Controls the offset of the added psuedo particles

Hardware Render Attributes:

Render Passes: Contols the number of times that a render is averaged

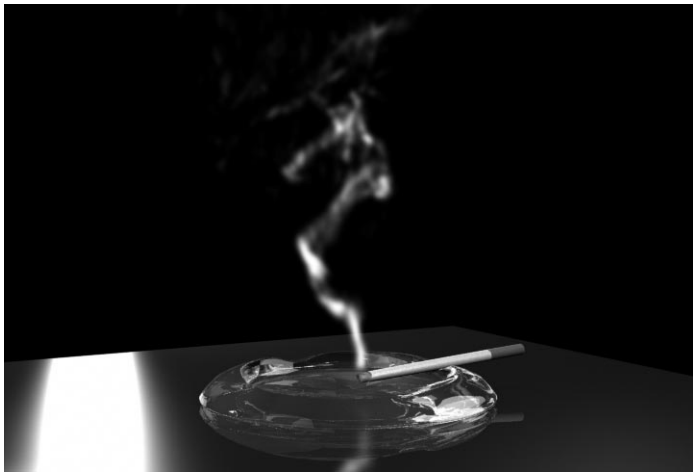
Edge Smoothing: Controls Anti Aliasing of geometry

Motion Blur: Controls samples of time taken to blur particles and geometry. Values and approach differ from Maya software motion blur.

Motion Blur and Caching

Motion blur allows you to average the look of the particles over time. This attribute is only available to hardware multi-pass render types. To use motion blur for multi-pass hardware render types you will also need to cache your particle motion.

Note: Several hardware configurations have know problems with using Hardware Multipass Motion Blur.



Example: Cigarette Smoke

In this example you will create and render particles for cigarette smoke using hardware rendering techniques. *cigSmoke.mb* contains props and dynamic elements ready for particle rendering. The animation of this setup is interesting as well. The main trick to the particle movement is two turbulence fields that have their phase animated with a simple sin expression. The only difference between the two fields is that the second turbulence field has a slightly slower frequency. These two fields work to reinforce each other while moving the particles. Rotation is obtained with a vortex field.

1 Open the scene file

This scene file consists of a cigarette and an ashtray and several fields.

- Open the file *cigSmoke.mb.*

  There is a parented emitter under the cigarette group. You will use this emitter as your source for the particle smoke.

  Playback the scene to get an idea of how the particles are moving.

2 Open the hardware render window

The Hardware Render Buffer is the name of this window. This window cannot be minimized.

- Select **Window** → **Rendering Windows** → **Hardware Render Buffer...**

3 Set Hardware Render Attributes

In the Hardware Render Buffer Window, Select **Render** → **Attributes...**, and set the following:

In the Image Output Files section:

Filename to **cigSmokeTest**

Extension to **name.0001**

Start Frame to **1**

End Frame to **100**

By Frame to **1**

Alpha Source to **Luminance**

In the Render Modes section:

Lighting Mode to **All Lights**

Draw Style to **Smooth Shaded**

Texturing to **On**

Line Smoothing to **On**

Full Image Resolution to **On**

In the Mult-Pass Options section:

Multi Pass Rendering to **On**

Render Passes to **9**

Motion Blur to **4**

4 Set the particle render type to multi-streak

This render type will provide streaking and multiple “jittered” psuedo particles.

- Select the *particleShape.*
- Change the **Particle Render Type** to **multi-streak.**

- Press **Add Attributes** For **Current Render Type,** and set the following:

  Depth Sort to **On**

  Color Accum to On

  Line Width to **1**

  Multi Count to **5**

  Multi Radius to **0.100**

  Normal Dir to **2**

  Tail Fade to **-0.500**

  Tail Size to **0.750**

  Use Lighting to **On**

5 Fine Tune lighting of particles

For hardware rendering you will find that lighting values are often different than what is appropriate for software rendering. You may want to use seperate lights for hardware rendering and seperate lights for software rendering.

- Select each light in the scene and duplicate it.
- Rename the light to a hardware designation
- Link the hardware lights to the particles and unlink the software lights to the particles.
- Light the particles with the hardware lights. Test render to the Hardware Render Buffer.
- Toggle **Color Accum** to see the effect that it has on the particle rendering.

  Color Accumulation: When transparent particles are infront of one another they can either render the nearest particle or add the overlapping particle's colors. By using color accumulation you can more closely simulate transparent particle effects. Note: some workstation system graphics do not support color accumulation through the accumulation buffer.

  Color Accumulation will also work to hide orphaned particles that are not contributing to the overall smoke trail. This will add to the smooth effect we are looking for.

6 Apply Caching to the particle objects

In order to apply hardware motion blur and render a sequence of images you will need to cache the simulation. Motion Blur requires knowing where the particle is and was, in order to determine the correct motion of the particle and determine the correct tail shape. Because future particle position is evaluated as the dynamic simulation calculates each frame, Maya will not be able to predetermine the motion blur future frames unless the calculations have already been performed and stored in memory.

- Select the particle objects.

- Select **Solvers** → **Scene Caching** → **Enable.**

  The particleShape will now show **Cache Data** attribute as **Enabled**

  To disable caching for a particular particle you can de-select this attribute.

Note: Each time you make a change to your scene, either to field values or *particleShape* attributes you will need to delete the cache.
To delete the cache you will need to select each affected particle and select **Solvers** → **Scene Caching** → **Delete.**

---

7 Add a shelf button to delete cache for your particles

Instead of having to select your particles and then select this nested solver menu it is much faster to take the command from the script editor and paste it to a shelf.

- Select your particles and delete cache.
- Open the script editor and drag the particle delete cache command to the shelf.

  ```
  particle -e -deleteCache particleShape1;
  ```

  Now you can delete your cache with a button click.

  If you have many particle objects you can combine these commands for a multiple cache deletion:

  ```
  particle -e -deleteCache particleShape1;
  particle -e -deleteCache particleShape2;
  particle -e -deleteCache particleShape3;
  ```

8 Adjust the motion blur

Experiment with different values of hardware Motion Blur.

- Open the Hardware Render Globals window.
- In the Multi-Pass Render Options section, set **Motion Blur** to values between **.5** and **6.**

  What do you notice about tail size and particle blending?

Note: Some hardware graphics configurations do not support hardware motion blur.

---

9 Adjust opacity over lifespan

The opacity of the particles should thin out as they get older.

- Add **lifespanPP** and **opacityPP** attributes and apply an opacity expression that is a function of age.

Grainyness is ok

Grainyness is to be expected in your final smoke renders. This will be smoothed and blurred during the compositing stage so you should not

work too hard to get rid of it. Anticipate that the small particles that are orphaned or not contributing greatly will also get removed during blurring and softening that takes place in your compositing software. Color accumulation will also work to remove or lesson errant particles.

What to do if you are running out of memory when caching

If you are doing this type of hardware render that requires caching,+ you may find that large scenes with many particles and many frames may be more information then your computer can safely store in memory. You may have to break up your scene into several frame ranges and render these frame ranges seperately with seperate cache runups.

To do this you will need to set the frame range, and cache **frames 1** to **155** then render **frames 1** to **150** then delete this cache, then select the new frame range of **frames 150** to **205**. Use **Solvers** → **Run-Up Caching** → **Run-Up and Cache** to run up to **frame 150** then cache **frames 150** to **205.** And so on...

The reason for the **5 frame overlap** is that the last 5 frames would have incorrect motion blur if you only cached up to 150 (1 to 150) the last 5 frames would not have this pre-calculation and thus the motion blur would be unpredictable.

This is anticipating a 5 frame motion blur setting. If your motion blur is for example only sampled from 2 frames then you would only need a 2 frame overlap. Another approach is to cache the extra frames (1 to 155) ahead of time but only render the frames up 150.

Optional Exercises:

- Render the fireworks from the emit lesson.
- Render the magic wand from the particle expressions lesson.

The emit lesson containes a very good fireworks simulation that needs some attention for rendering. Use the *emitFinal.mb* scene to work on using **multi-point** and **multi-streak** with Motion Blur.

The *magicWand.mb* scene is another scene that could use some fine tuning for hardware render.

SOFTWARE RENDERING

In this exercise you will software render the raygun scene with software particle effects. You will build a shader for the particles and apply this shader to the particles. Software rendering of particles will allow you to do post process effects such as glow and incandesence as well as interactive effects of reflection, object oclusion and shadows. The price for this functionality is time.

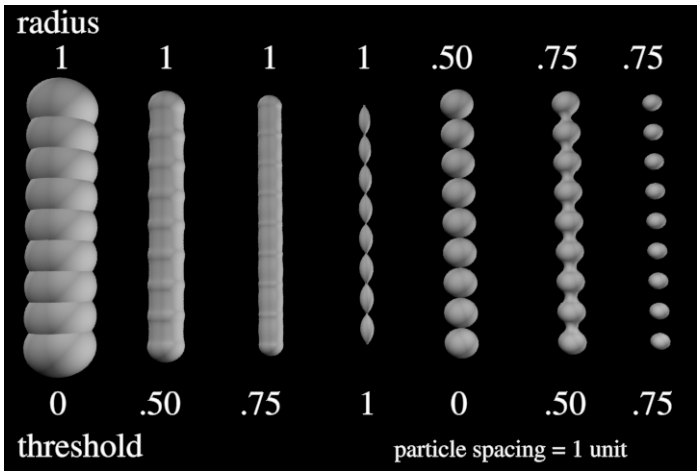
There are three types of software particle render types. They each serve a seperate purpose but can be combined as well.

Blobby Particles

Blobby particles depend on each other to form blobs or connected shapes based on their radius and proximity to each other. There are two attributes related to the blobby particle object that controls this behavior.

Radius: Sets the diameter of the blob.

Threshold: Sets the amount of flow between adjacent particles. The resultant blobbyness is a function of the particle radius, threshold and distance between particles. All of these factors are interdependant so experimentation is almost always necessary.



Blobby particle comparison of radius vrs threshold

To see a comparison of Radius vrs. Threshold open the scene file *blobCompare.mb*. Note that **threshold** values that are much larger then the **radius** may result in invisible particles.

Blobby Particles are also the only render type to which you can assign shading groups and render in the same way you would assign shading groups to NURBS or polygonal geometry.

It is generally recommended that you avoid depending on 2d mapping on blobby particles as blobbies use an averaging of UV space to calculate their respective mapping coordinates. This can result in artifacts or incorrect mapping of the blobby surface. 3D mapping works the same as you would expect for any surface.

Cloud Particles

Cloud Particle Render type, as its name implies, is designed to create volumetric rendering effects. It has an additional attribute called Surface Shading which controls how much blob the cloud will have.

Radius: Works much like the Blobby Surface attribute

Threshold: Works much like the Blobby Surface attribute

Surface Shading: Adjusts the degree of Surface shading applied to the particleCloud shader via the Surface Shader input on the Shading Group.

Tube Particles

Tube Particles are the software counter part to the hardware streak particle render type. The Tube particle has attributes to control the size of either end of the particle as well as tube length. The Tube Particle type also does not have the Surface Shading capability that the Cloud and Blobby type have. The Tube Particle is Velocity dependant like the Hardware streak types. The tail length and direction is dependant on the velocity and direction of the particle.

Radius 0: Controls the tail size.

Radius 1: Controls the head size.

Tail Size: Controls the tail length but is also proportional to the particle velocity.

Particle Utility Nodes

The Particle Mapping Utilities provide a method of transfering particle object and per particle array information to the shader. This way you can control the look of a particle in relation to particle and per particle attributes such as opacity or opacityPP.

- Particle Age Mapper
- Particle Color Mapper
- Particle Incandescence Mapper
- Partcle Transparency Mapper

The particleCloud shader provides lifespan based mapping for Color, Transparency and Incandecsence with the Particle Age Mapper node. You can also use the Particle Mapping Utility nodes (Particle Color Mapper, Particle Transparency and Particle Incandescence Mapper) to map dynamic attribute values into a shader from the particleShape that are not necessarily age based.

Shading Group Organization

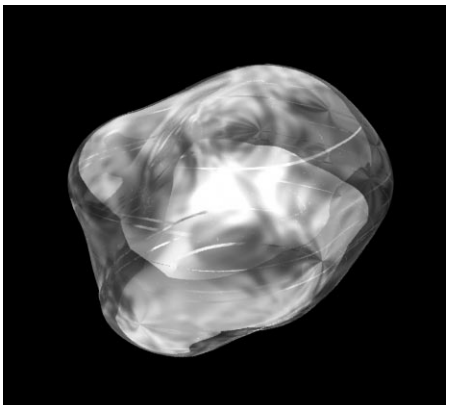
The Maya rendering engine accepts three basic types of shader information. Surface, Volumetric, and Displacement. These three types of rendering information are passed to the Rendering Partition through the Shading Group node. The Shading Group node acts as a place holder that tells the Rendering Partition what will be rendered and which shaders are to be used on which objects. The Light Linking Partition also looks to the Shading Group to determine which lights will work with which shaders and / or objects.

Software Particle rendering makes use of the Surface and Volume shader inputs to the Shading Group node. You connect Surface shaders (materials) and Volumetric shaders (particleCloud) to the three different types of particle rendertypes via the particleCloud Shading Group (particleCloudSG). Surface shading is plugged into the Surface Material input of the Shading Group and Volumetric shading (particleCloud) is plugged into the Volume Material input.

Blobby particles make use of the standard surface material shaders such as anisotropic, phong, lambert etc...

Cloud particles make use of the surface materials and the volumetric partricleCloud shader by plugging materials into the Surface Material input of the Shading Group and the particleCloud shader is plugged into the Volume Material input.

Tube particles make use of the volumetric shader particleCloud only. ParticleCloud shader is plugged into the Volume Material input of the Shading Group.



Blobby Particles with Environment Reflection Map

Example: Space Blob

1 Open scene file

- Open the file *spaceBlob.mb*

  This scene consists of particles orbiting a newton field.

2 Set the Particle Render Type to Blobby particles

3 Apply phong shader to the blobby particles

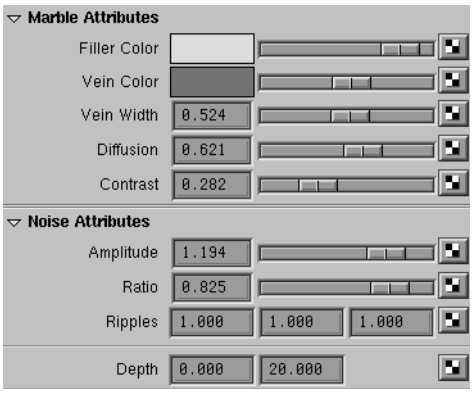
You will use a phongE material as the basis for the blobby particle shader. From this material you will add other rendering nodes to control color, reflection, and specularity.

- In the HyperShade window, **RMB** select **Create** → **Materials** → **PhongE.**
- **LMB** press to deposit the node in the appropriate place in the hypershade window.
- **Shift-select** the particles then the phongE node, **RMB** on the phongE node to select **Assign Material** to selection.

4 Create the color and specular map

Because you want the blob to mimic clear water like material, it will get most of its color from its environment. But you will still want some method to control the color and specularity. The 3d texture Marble makes a good sky texture.

- Press the **Map** button for **Color** in the Attribute Editor.
- In the Create Render Node window, press **Marble** under the 3D Textures section.
- Below are some appropriate values:

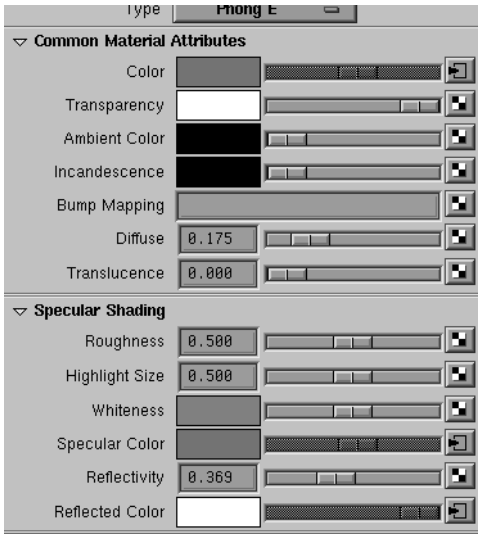


- Rename the marble node to *skyMarble.*
- **MMB-drag** *skyMarble* from the hyperShade window to the **Specular Color** input on the phongE attribute editor.

5 Add environment map to the reflected color of the phong

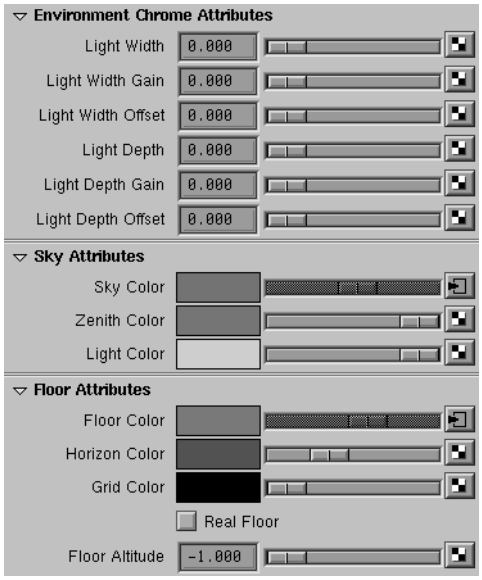
Environment maps are a quick method to mimic ray traced reflections and refraction.

- Set the following values for the phongE material node:



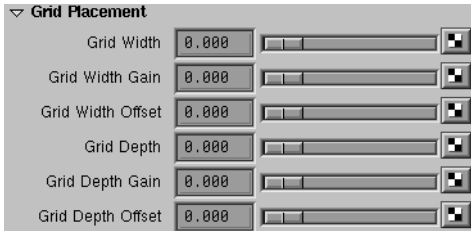
PhongE node settings

- Map the **Reflected Color** attribute with an **Env Chrome** Environment texture**.**
- Set the following light values for the chromeEnv node to **0**.



Settings for the envChrome node

- Set the grid attributes to **0**.

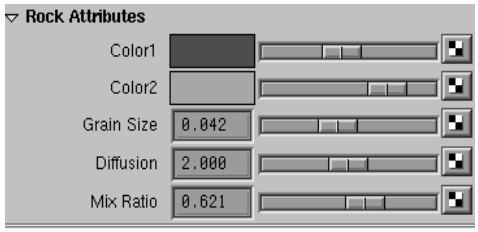


Grid settings

6 Map 3D textures into the chromeEnvironment inputs

The Chrome Environment map has several inputs for Sky and Floor color values. You will create a new texture for the floor and share your *marbleSky* texture for the sky color.

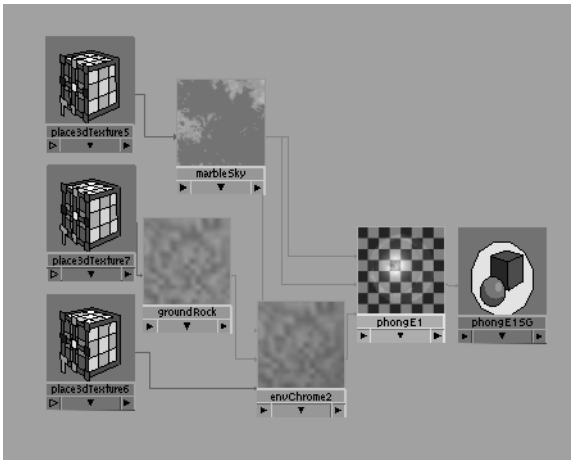
- **Map** the **Floor Color** with a **Rock** 3D texture.
- Set color values to approximate what your ground will look like.



Rock color

- From the Hypershade window, **MMB-drag** the *marbleSky* texture onto the **Sky Color** input of the envChrome node.

  Your shading network will look something like this:



Blobby Environment Shader network

7 Create a stand in sphere to IPR render with

Maya's IPR rendering workflow does not incorporate particles but you can create a stand in object to apply the shader to. After , you will apply this shader to the blobs. IPR is a much quicker way to do the fine tuning of a shader.

8 Adjust values to taste

particleCloud shader

The particleCloud shader provides tools for many particle rendering effects. This is the most flexible shader mechanism for software particle rendering. The particleCloud shader provides volumetric density control in addition to surface material attributes like color, transparency, glow and incandescence.

Density and Transparency

Density is closely related to transparency and will interact with the method you choose to drive the particle opacity. The **Density** Attributes allow you to control how a particle will look at it's edges and where it overlaps with other particles. Density can be thought of as the **volume transparency** or as another series of inputs that add greater control over several aspects of shader transparency. It is important to understand how Density works as the Transparency attribute alone will only get you part of the look you may be looking for. The **Transparency** attribute will work as the base of particle transparency where as the Density attributes apply to the volumetric portion of the shading. The third component is the **Blob Map** which allows a texture to be added to the internal structure of the particles appearance. Typically, the less **Blob Map** that is applied, the less of the particle surface outline is visible.

Density: This attribute controls how dense the particle shading appears inside the volume. This attribute is also necessary to "beef up" the shading if there is alot of noise, for example. Or if you are using the transparency attribute to drive the surface

shading you may find that the Density attribute can control the Volume portion independantly. Typical values between (0-1) control slight density but values up to 10 may create interesting internal density for a very transparent particle.

Noise: This attribute controls the amount of noise applied to the mapping of the density. Noise can give you control of how extreme the mapping will diffuse across the particle. Typical values can range from (0–4) but will also depend on how much density is being used.

Noise Freq: This attribute controls how large or small the spacing of the Noise maps across the particle. Typical values can be quite small (.01–.1) for low frequency noise.

Noise Aspect: This attribute controls at what angle or "shear" the noise will appear to exist in the volume. Typical values of (-1– 2) will change the noise direction from horizontal to vertical.

Blob Map: This attribute controls the mixture of surface shading and volume shading independant of the presence of a Surface Material. It can have a texture applied through it to create more interesting internal structure to the particle appearance. It will interact with the overall density and transparency. If you turn this value very low you may need to increase the density and / or noise, and / or transparency to see the cloud particles. A minimum value above 0 for Blob Map is necessary for particles being rendered with no Surface Material Shading. Otherwise they will be invisible. Blob Map is a scaling factor for transparency.

Life Mapping

When you map a texture into the Color, Transparency or Incandescence, Maya will also map this shader into the appropriate "Life" mapper attribute if there is a lifespan attribute on the particleShape. The particle will have this color or transparency or incandescence for its entire life.

If you map a texture into the "Life" mapper it will override the texture setting in the appropriate attribute and apply this new texture as the mapped color. This "Life" mapped texture will then provide the particle color, transparency or incandescence for its current age throughout it's lifespan by the addition of the Age Mapper utility node. This node is created and connected automatically. The current color is mapped to the current age based on the mapping coordinates of either U or V of the texture. The Age Mapper utility node thus converts the particle age to the appropriate value along the U or V texture coordinates and supplies this information to the renderer to determine the particle color at that moment.

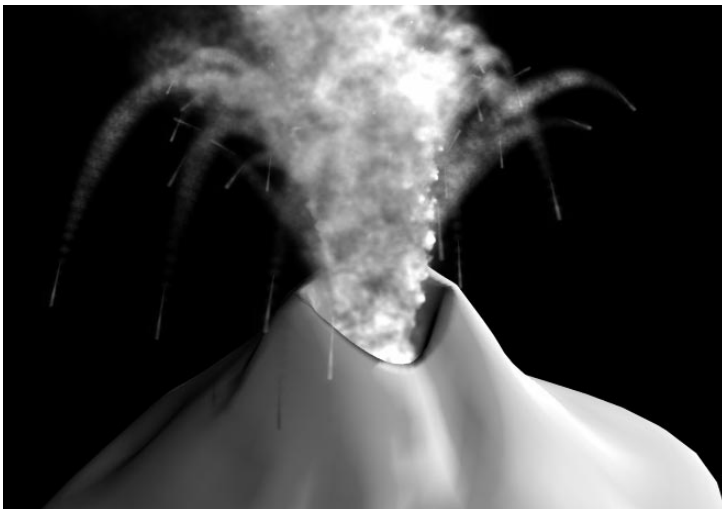
Age Mapper Attributes: The relative age flag of the Age Mapper node determines whether or not the particle will spend it's entire lifespan going through the texture or if it can run through the texture more than once during it's lifespan. The Time Scale multiplier controls how much of the texture is covered per frame

of age and Fold At End controls if the particle can go back through the texture with Relative Age off - Reversing the color direction at the end.

Self Shadowing

Self Shadowing is an important part of getting realistic cloud-like particle rendering. To self shadow particles you must:

- **Ray Trace** render the scene.
- The appropriate lights must be set to cast **Ray Trace shadows**.
- The *particleShape* attributes for **Better Illumination** and under Render Stats **Casts Shadows** should be set.



Example: Volcano

1 Open the scene file

This scene consists of a mountain with a hole in the top. Particles are emitting from the crater. There are 3 groups of particles - cloud particles, ejecta particles and thick cloud particles.

- Open the file *volcano.mb.*

  The seperate particle elements have been put on seperate layers to facilitate their visibility and selection. Use these layers to operate on only one particle layer at a time. Otherwise this can be a large and un-manageable scene due to the large number of particles. Also keep track of the render globals to keep render times down.

2 Set Particle Shading passes to 1

- Open the Render Globals window, and then open the Anti-aliasing Quality section.
- In the Number of Samples section, set **Particles** to **1**.
- Toggle off **Better Illumination** for each particle object during rough initial testing.

Ray tracing should be done from the beginning because lighting and illumination are considerably different from non raytraced rendering. For this reason it is adviseble to find as many ways as possible to reduce the load on the renderer using Layers to control visible geometry during testing for example.

Ray tracing provides the best self shadowing of particles. To speed rendering during testing use only the most course values for shadow quality. The default values should be adequate for this.

3 Create the cloud material for the cloud particles

- In the HyperShade select **Create** → **Materials** → **Volumetric** → **Particle Cloud.**
- Rename the *particleCloud* material and its shading group to *cloudShader* and *cloudSG* respectively.

  Note that the cloudSG has cloudShader as its Volume Material input.

Note: **With Shading Group** should be **On** in the Create Render Node window... this will automatically create a shading group when a material is created.

---

4 Assign the cloud shader to the cloud particles

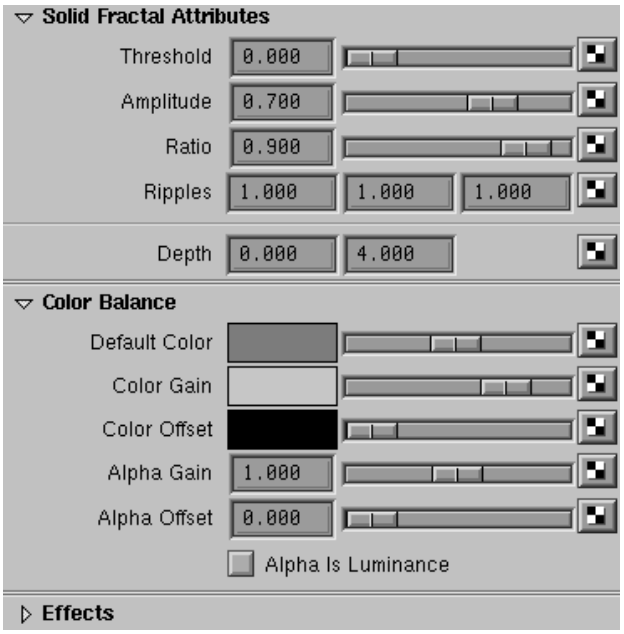
- **Shift-select** the cloud particles in the viewport and the *cloudShader* material in the Hypershade window.
- With **RMB** over the material node in the Hypershade, select **Assign Material to Selection** from the pop-up menu.

5 Map Solid Fractal textures to color and transparency

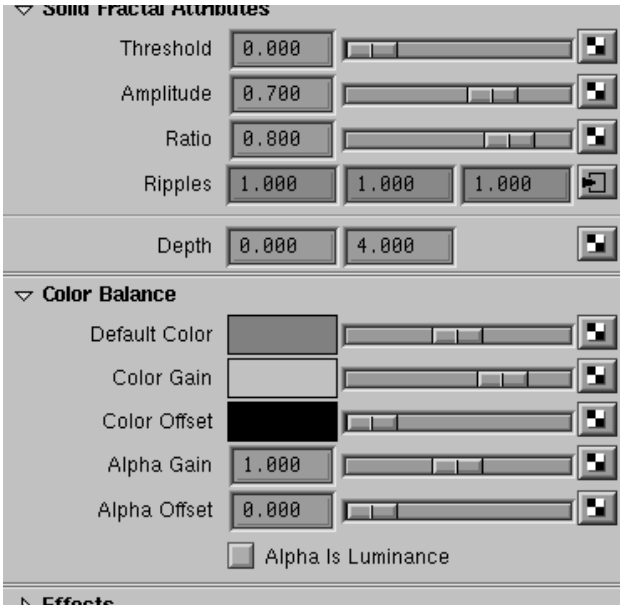
- Map the *cloudShader* **Color** attribute with a **Solid Fractal** 3D Texture.
- Repeat this procedure for the *cloudShader* **Transparency** attribute.
- Rename these textures as *colorFractal* and *transFractal*.

6 Adjust Fractal texture parameters

The Solid Fractal textures will need the attributes set to create the correct scale of smoke. The Color Fractal and the Transparency Fractal will need to be adjusted equally so that they track with each other. Color Gain attributes on the textures will control much of the strength of the Brightness and Transparency.



Color Fractal Settings

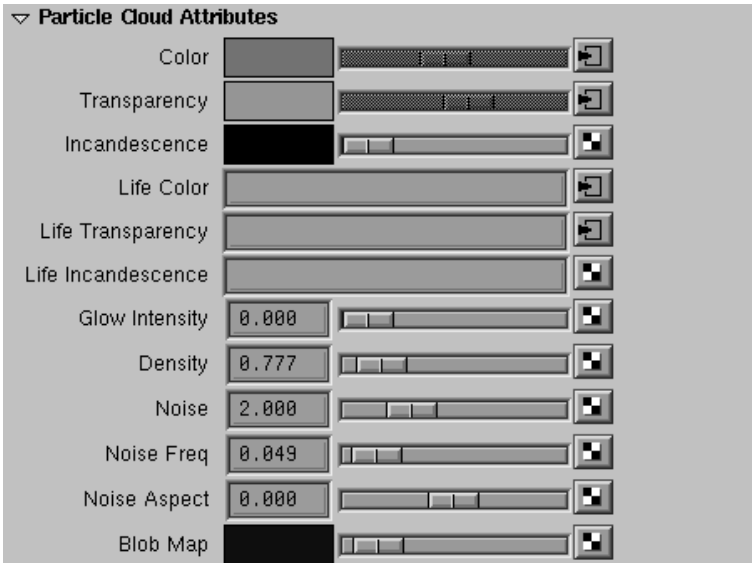


Transparency Fractal Settings

Tip: For attributes like Ripples that you will want to track equally on the two Fractal textures you can connect them to each other with the connection editor. This way when you change one texture the other will follow.

7 Adjust cloudShader attributes to achieve a smoke like appearence

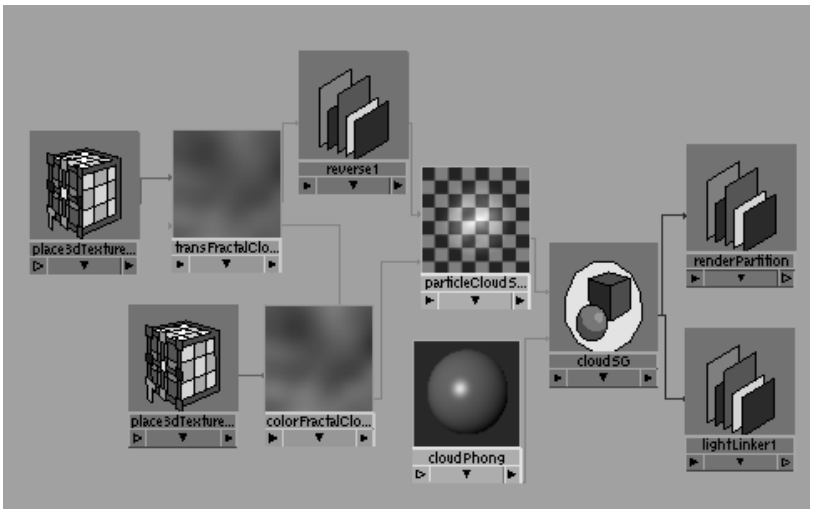
The **Transparency** and **Density** are the most important part of getting a soft volumous look. This also requires the most tweaking. Use the values below as a starting point. Note how low the **Blob Map** is set.



8 Use a Reverse Utility Node to invert the Fractal Transparency

You may find that the Transparency Fractal is not tracking the way you expect it to. As you increase its buildup of luminance to match the color fractal, the transparency fractal will create more transparency. By using a Reverse Utility node between the Transparency Fractal and the *cloudShader* transparency input you will get the effect of more luminance and less transparency, like you see in real clouds.

- Display the *cloudShader* network in the HyperShade window.
- With **RMB** select **Create** → **Utilities** → **General** → **Reverse** then **LMB** in the HyperShade window to deposit the Reverse node.
- Connect the **OutColor** of the *TransFractal* texture to the Input of the *Reverse* node. Connect the Output of the *Reverse* node to the **Transparency** input of the *cloudShader*.



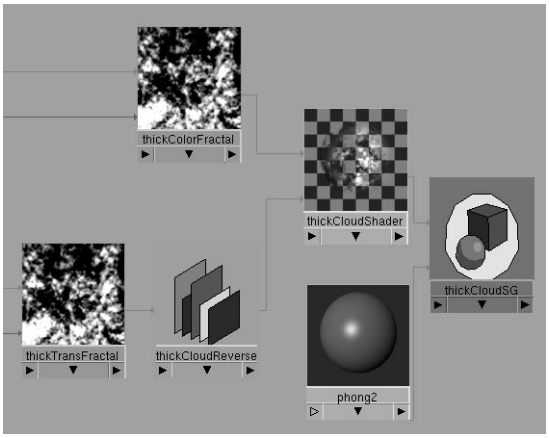
Graph of cloudShader network

9 Create the thick cloud shader for the thick cloud particles

Create the thick cloud shader from the *cloudShader* you just created. For this shader you will simply modify the attributes to create a thicker and more turbulent look.

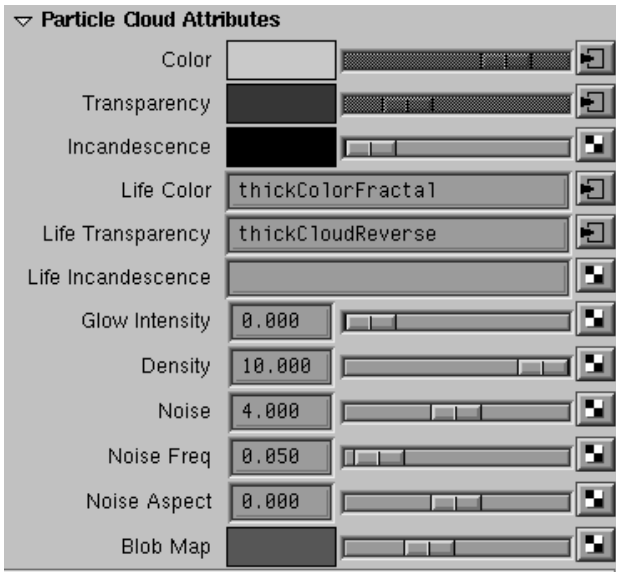
- Select the *cloudShaderSG* shading group in the HyperShade window.
- With **RMB,** Select **Edit** → **Duplicate** → **Shading Network.**

  This will duplicate the upstream nodes and connections as well as the selected shading group.
- Rename these nodes with the "*thick*" prefix.

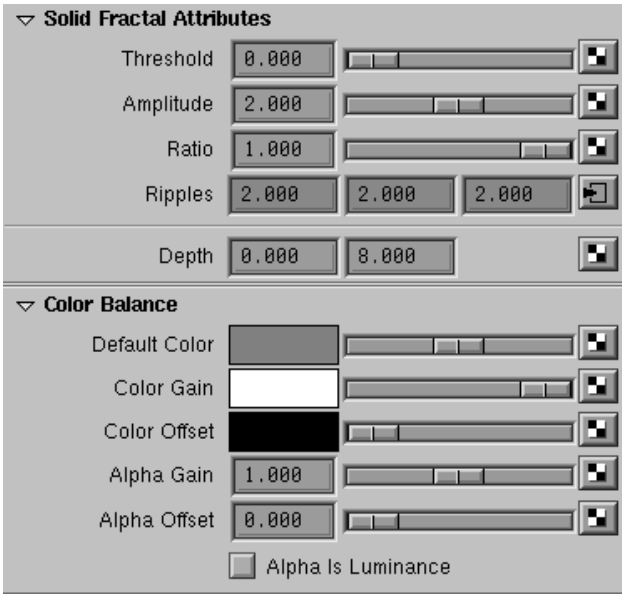


HyperShade Graph of thickCloud shader

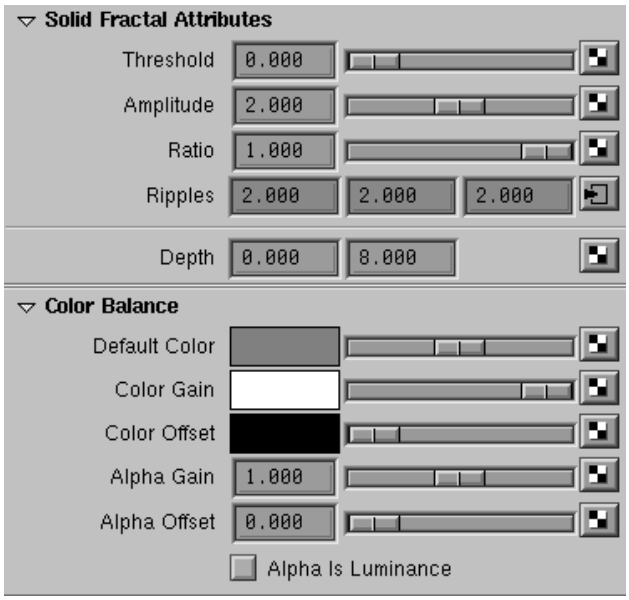
- Apply this material to the *thickCloud* particles.
- Use the following diagrams as guides for setting the Fractal and Material attributes.



thickCloudShader attribute settings



thickColorFractal texture attribute settings



thickTranFractal texture attribute settings

10 Create the ejecta shader for the tube particles

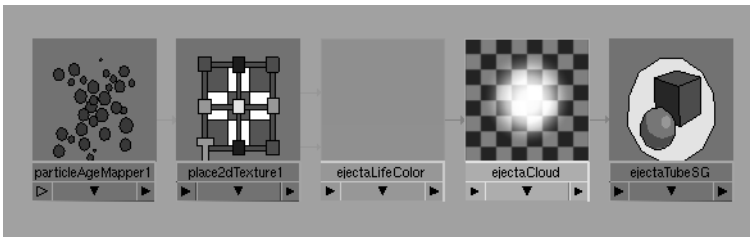
For the streaks of red hot rock being ejected from the crater you will use the Tube particle type.

- In the HyperShade window, **RMB** select **Create** → **Material** → **Volumetric** → **particleCloud.**
- Rename this material and shading group as *ejectaCloud* and *ejectaTubeSG.*
- Assign this shader to the *ejecta* particles.

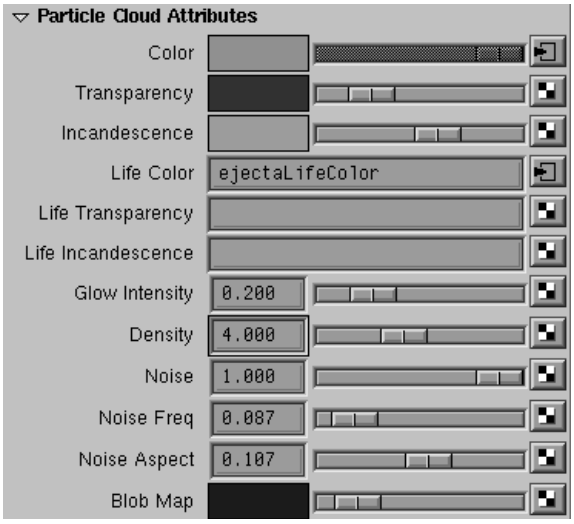
11 Create a Life Color Ramp texture

- Press the **Map** button for **Life Color** for the *ejectaCloud* material.
- Press **Ramp** from The Create Render Node window.

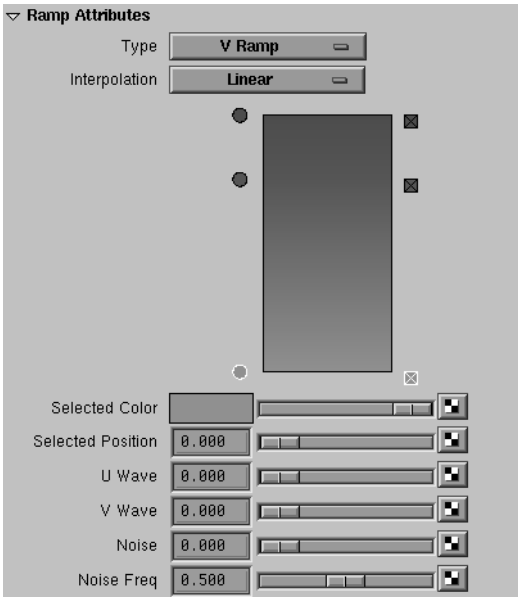
  Because their is a lifespan attribute associated with the ejecta particles this ramp texture will now control what the particle color is throughout its life. A Particle Age Mapper node has been created feeding into the 2D texture placement node. This Age Mapper will convert the age of the particle into a value that is referenced against the V direction of the Ramp texture. The color found at that value is then fed to the color input of the particle cloud material thus determining the particles color at that moment.
- Use the following illustrations as a guide for setting attributes on the *ejectaCloud* material and the *ejectaLifeColor* ramp.



Graph of ejecta particle shader with Life Color Ramp and Particle Age Mapper



ejecta particle shader settings



ejectaLifeColor ramp settings

Exercises:

- Add a smoke trail to the ejecta particles.

- Add lava to the volcano experience with blobby particles.

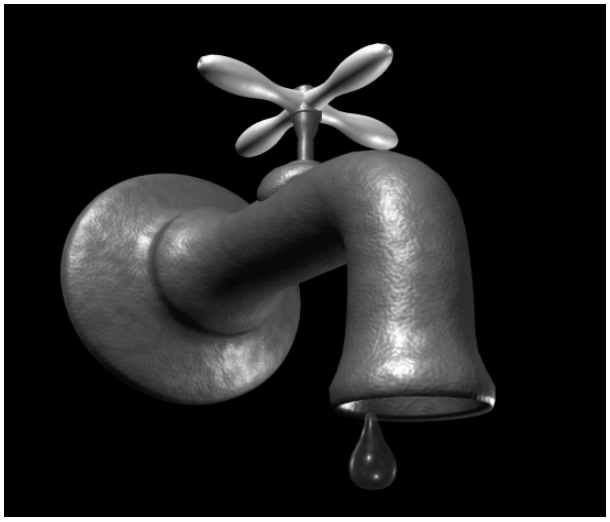
SUMMARY

- Hardware Rendered Particle Types
  - Point
  - Multipoint
  - Streak
  - Multistreak
  - sphere
  - sprite
- Cigarette Smoke technique using motion blur
- Caching and Runup and cache
- Software Rendered Particle Types
  - Blobby
  - Cloud
  - Tube
- Shading Group organization
  - Surface Material
  - Volume Material
  - Displacement Material
- Blobby particle techniques
- Environment Mapping
- Smoke and Cloud Techniques
- Density and Transparency
- Fractal 3d texture

13 Soft Bodies

Controlling Surfaces with dynamic motion is accomplished using Soft Bodies. In this Lesson you will learn the following:

- Creating Soft Bodies
- Soft Body parameters
- Particle Springs
- Soft Body application
- Goal weighting with Artisan



Chapter 13

SOFT BODIES

When you make geometry or a lattice a soft body, Maya creates a corresponding particle object. The particle object is placed in relation to the surface based on options that you select when you create the soft body. A duplicate surface can be created and used as a goal object to help control the soft body and maintain it's shape.

NURBs and Polygonal surfaces can be made into soft bodies. The particles are created and placed at the corresponding CVs or Vertices.

Dynamic Springs can also be applied to the particles to control the tension that exists between the particles and thus the CVs or Vertices.

Creating Soft Bodies

Creating a soft body object involves selecting the geometry to be made a soft body then selecting from the Dynamics menus, **Bodies** → **Create Soft Body -** ❐.

The Soft Body creation options allow you to make the selected object a soft body in the following manner:

Make Soft: This option directly makes the selected object a soft body.

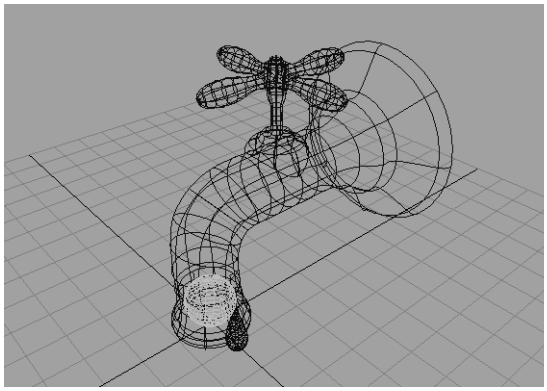
Duplicate Make Copy Soft: This option duplicates the object and makes this duplicate the soft body. This is useful for making the original object a goal for the soft body.

Duplicate Make Original Soft: This option duplicates the object in question but makes the original selected object the soft body. This option enables the creation of a duplicate for use as a goal object.

If you are duplicating you can choose to **duplicate the upstream graph** of the selected object as well as automatically **hiding** or making a **goal** of the **Non-Soft objects**.

Example: Faucet Drips

In this example you will create drips for a faucet spout.



1 Open the scene file

- Open the file *faucet.mb*

  This scene file consists of a faucet and some spheres for use as dripping water targets.

2 Make drip into a soft body

- Select the drip object.
- Select **Bodies** → **Create Soft Body -** ❐, and set the following:

  Creation Option to **Make Soft**.
- Press **Create.**

  This will turn the drip object into a soft body. Note the *dripParticle* object that is created under the drip object.

3 Add drip and drop targets as goals

- **Shift-select** the *dripParticle* object then the *dripTarget* object.
- Select **Particles** → **Add Goal** to create a goal object for *dripParticle.*
- Repeat this for the *dropTarget*.

4 Playback the scene

Note that the drip object assumes a location between the goal objects.

5 Create an emitter for the drip soft body

Maya enables you to emit soft body particles from an emitter. To do this you will create an emitter and connect the soft body particles to this emitter.

- **Particles** → **Create Emitter -** ❐

  Create a **Directional** emitter and set its **Rate** to **300.**

  After you create the emitter you can delete the particle object that was created with the emitter.
- Rename this emitter *dripEmitter.*

6 Connect dripParticle to dripEmitter

- **Shift** or **Cntrl-select** the *dripParticle* object then the *dripEmitter* object.
- Select **Connect/Add** → **Connect to Emitter**

  This will connect the *dripParticles* to be emitted from the *dripEmitter*.

7 Turn off “Enforce Count From History”

On the *dripParticleShape* there is an attributes section called Soft Body Attributes. These are the attributes that control soft body specific behavior of the particles.

- Turn **off** the **Enforce Count From History** attribute.

  This will allow the soft body to exist without all it’s particles while it is being emitted. In other words, before all it’s particles have been born.

8 Set Max particle count to the number of softbody particles

You don't need more particles in the *dripParticle* object then the number of particles needed to satisfy the softbody.

- Set **Max Count** equal to **Count,** which in this case is **56.**

9 Create an expression for target goal weights

By setting your playback range to 30 frames and playing back you can get a feel for what the goal weights should be set to for the drip to progress from the drip to the drop.

- While playing back from the timeline adjust the goal weights of the *dripTarget* and *dropTarget* to simulate the drip falling under the influence of each in succession.
- Under the **goalPP** attribute for *dripParticleShape* enter the following as a creation rule:

  ```
  dripParticleShape.goalPP = 1;
  ```

  This expression sets the particle goalPP value to 1 at particle birth.
- Under the **goalPP** attribute for *dripParticleShape* enter the following as a runtime rule:

  ```
  dripParticleShape.goalWeight[0] =
  1-(linstep(5,40,frame));
  dripParticleShape.goalWeight[1] =
  linstep(0,40,frame);
  ```

  These two expressions control the goal weight attributes and apply linstep functions to each. Goal weight attribute values are stored in an array called `goalWeight[]`. The goal weights are then accessed in order of how they were created.

10 Playback the scene

You should see the particle progress from the drip to the drop targets. Work with the goalWeight expressions to fine tune this motion.

Goal smoothness will also play a part in the motion of the drip. A value of **6** may suffice.

11 Add Gravity to the dripParticle

The dripParticle is behaving linearly towards it's goal targets but has no life, bounce or globbyness. We can add fields to the particles to get them to act a little less orderly.

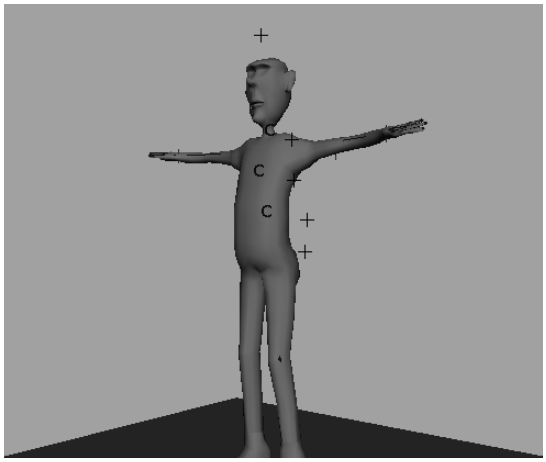
- Select the *dripParticle* object and select **Fields** → **Create Gravity**.

  A large **magnitude** may be necessary to influence the particles above the goal weight dominance. Try a value of **50**.

12 Playback the scene and tune as necessary

Example: Leroy Lalane

In this exercise you will turn an animated character into a soft body then use Artisan to set the goalPP weights on parts of the character that you want to react dynamically.



PolyCharacter Leroy

1 Open the scene file

This file contains a polygonal character that is animated.

- Open the file *leroy.mb*

  The character has been setup with a skeleton and skinned. Keyframe animation has been used to animate the joints and locators that are point constrained to the IK handles. The main group, *LeroyAll* contains two groups one called *leroySetup* which contains the skeleton and control components and another object which is called *Skin*, this is the geometry that is bound to the skeleton.

2 Duplicate the character Skin with Inputs

In order to create a soft body character you will need two skin objects. One to act as the soft body and the other to act as a goal to the soft body. Both of these will be bound to the skeleton.

- Select *Skin* object.
- Select **Edit** → **Duplicate** - ❐, press **Reset** and set the following:

  Duplicate Input Connections to **On**

- Press **Duplicate.**

  A new Skin object will be created that has the same incoming connection from the bind skin process which binds the geometry to the skeleton.

- Rename the new object from *Skin1* to *goalSkin*.

3 Make the Skin object a soft body

- Select the *Skin* object
- Select **Bodies** → **Create Soft Body -** ❐, and set the following:

  Creation Options to **Make Soft**
- Press **Create.**

  This will create a soft body of the selected object without creating duplicates and handling the target goal and goal object visibility. You will be doing this manually.

4 Make the goalSkin a goal for the SkinParticle soft body

- **Cntrl-select** the *SkinParticle* object that is under the *Skin* object and then the *goalSkin* object in the Outliner.
- Select **Particles** → **Add Goal.**

5 Paint the goalPP weights for the newSkin soft body

By **RMB** selecting over the character you will access the paint tools available for operation on the character.

Marking Menu paint Options:

SkinShape : rgb

SkinParticleShape : goalPP, opacityPP, rgbPP

SkinCluster : skinWeights

- Select **goalPP** under the **RMB** marking menu **paint** → **SkinParticleShape** → **goalPP.**

  This will invoke the Attribute Paint Tool
- **MMB** drag the attribute paint tool icon from the **last picked tool** panel to your **Artisan Shelf**.

  By dragging this icon to your shelf you will be able to double click on this icon to bring up the options panel for this tool.
- In the Attribute Paint Tool options window set the following options prior to painting **goalPP** weights:

  Radius (U,L) to **2**

  Opacity to **1**

  Value to **.25**

  Operation to **Scale**

  Paint Offset to **0**

  Paint Mult to **1**

  Clamping to **Both**

  Clamp Lower to **0**

  Clamp Upper to **1**
- Paint on the Leroy's Skin in areas where you want to scale the goalPP value by 1/4 percent on each stroke.
- Playback to confirm that the soft body is reacting accordingly.

You can also select the appropriate particles and load them into the component spreadsheet.

Tip: To erase and start from scratch, set **Operation** to **Replace** then set the **Value** to **1** and press **Flood**. This will flood the surface with one value of the selected attribute.

---

6 Add Springs to the Particles of the soft body

To bring the particles under some control and resilience, you will add some springs to the particles that are moving around too much.

- In component mode select the particles that are around Leroy's stomach.

  A quick way to do this is to select the soft body object in the perspective view then **RMB** to select **Particle** from the pop-up menu, then drag select the proper particle components.

- Select **Bodies** → **Create Springs** - ❐, press **Reset** and set the following:

  Creation Method to **Wireframe**

  Wire Walk Length to **1**

- Press **Create.**
- Playback animation
- Experiment with different stiffness and damping values.

SPRINGS

Springs and Soft bodies often work hand and hand. You will often find that a goal object is not always appropriate for controlling a soft body that is deforming or colliding with another object.

- Springs are useful for controlling particles that will respond to forces with cohesion.
- Springs can be established between particles.
- Springs can be established between particles and surface CVs or polygonal vertices.
- Springs can be established between surface CVs and or polygonal vertices.

Springs often require **Stiffness** settings above **100**. Springs may gain little benefit from higher **Damping** values though. Experimentation is required for each application but if you find that a higher Damping value results in poorer spring response, your object may require more springs or you may need to increase the **oversampling** of the simulation.

Overlapping springs and springs connected in multiple directions will have profound results on the simulation. Experimentation is required.

Adding and removing springs from a spring object

The scene file *springCompare.mb* contains three examples of soft body spring approaches. They are all under a gravity field and will collide with the floor.

- The first sphere is a soft body with no springs.
- The second example is a sphere with a wrap deformer which is a soft body with springs.
- The third example is a sphere with a lattice deformer which has been made into a soft body and springs applied.

When you playback this scene file you will notice that only the lattice sphere is resisting the force of the floor when it collides. This is because only the lattice example has springs in place positioned to oppose this force.

There are several methods to create or add springs to a spring object.

WireFrame walk length - Adds springs along the wireframe segment as determined by the walk length.

All - Adds springs between all the components that are selected.

Min/Max - Adds springs between the selected components that fall between the Min / Max criteria.

Set Exclusive - Is used to make the spring creation take place between objects and not between the components on the object.

1 Open the scene file

- Open the file *springCompare.mb.*

2 Add Springs to the WrapSphere's Spring object using wire walk length

- Select the *wrapSphere's* particles from the viewport in component mode.
- **Cntrl-select** the wrapsphere's spring object in the Outliner or **Shift -select** in the Hypergraph.
- Select **Bodies** → **Create Springs -** ❐, and set the following:

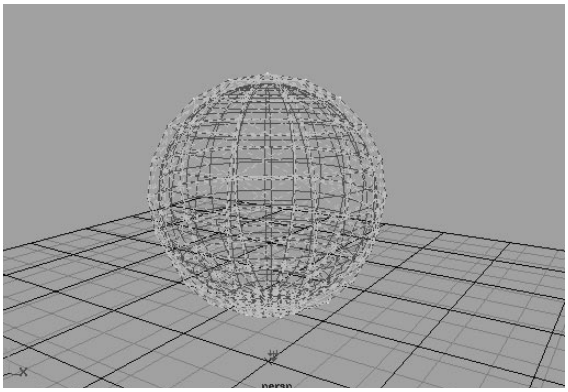
  Add to Existing Spring to **On**

  Don't Duplicate Springs to **On**

  Creation Method to **WireFrame**

  Wire Walk Length to **2**

- Press **Create.**



Wrap Sphere with WireFrame Walk Length of 2

3 Play back the animation

Note the sphere still collapses. There are still no springs directly opposing the particles journey to the floor.

4 Add Springs to the WrapSphere's Spring object using selected particles

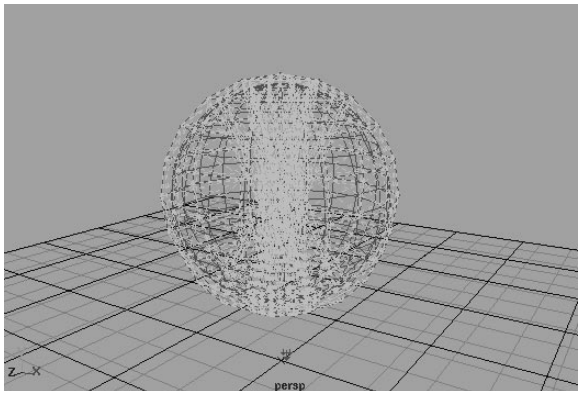
- **Shift-select** the *wrapSphere's* particle components that lie on the bottom of the sphere and the top of the sphere from the viewport in component mode.
- **Cntrl-select** the *wrapsphere's* spring object in the Outliner or **Shift -select** in the Hypergraph.
- Select **Bodies** → **Create Springs** - ❐, and set the following:

  Add to Existing Spring to **On**

  Don't Duplicate Springs to **On**

  Creation Method to **All**

- Press **Create.**



5 Add Springs to the WrapSphere's Spring object using Min/Max

When you find that selecting all the components and manually building springs would be too tedious, you may find that you can use the Min / Max criteria to distribute the springs in the right quantity.

- Undo the last group of springs that where created in the above step **or** select them in the viewport and delete them by pressing backspace.
- **Shift-select** all of the *wrapSphere's* particle components.
- **Cntrl-select** the *wrapsphere's* spring object in the Outliner or **Shift - select** in the Hypergraph.
- Select **Bodies** → **Create Springs** - ❐, and set the following:

  Add to Existing Spring to **On**

  Don't Duplicate Springs to **On**

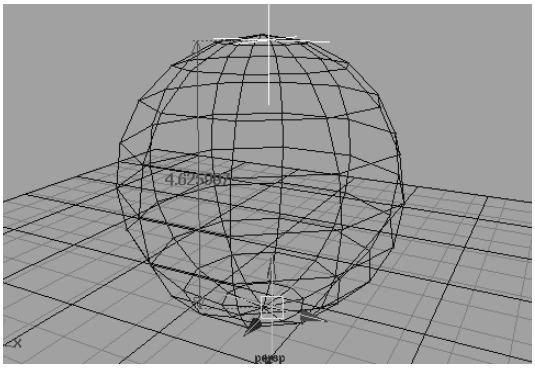
  Creation Method to **Min/Max**

  Min Distance to **4.6**

  Max Distance to **8**

- Press **Create.**

  You may find that trial and error are the best method to determine what are min and max values that will work best. Another method is to use the **Modify** → **Measure** → **Distance Tool.**

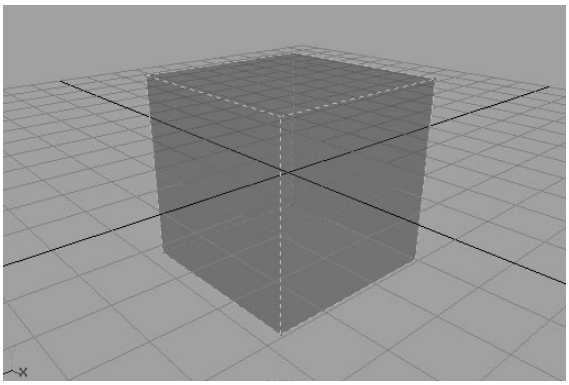


Using the Distance Tool

Working with Per Spring Attributes

The three Spring object attributes **StiffnessPS, DampingPS, RestLengthPS** are per-spring attributes. These attributes can have values set for individual springs through the Component Editor**.**

The scene file *springCube.mb* contains a polygonal cube that is 4 units in width, height, and length. This cube has springs applied to it by one wire walk length method.



Soft Body cube with springs at each corner.

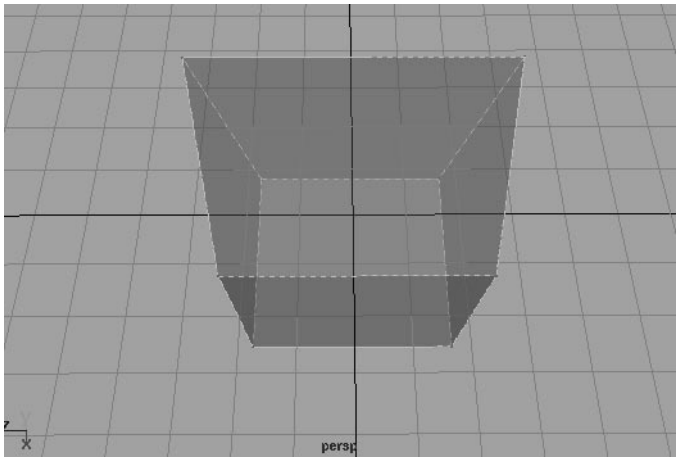
1 Open the scene file

- Open the file *springCube.mb*

  Note the spring object **restLength** size is **4**.

2 Change the restLengthPS on one of the cube's springs

- Turn **On** use **RestLengthPS.**
- In component mode, select one of the Cube's springs.
- Open the Component Editor and press **Load Components**
- Enter a value of **2** for the **restLengthPS** attribute.
- Playback
- Enter a value of **6** for the **restLengthPS** attribute
- Playback



Spring restLengthPS set to 6

3 Change the value of the spring object's End1 Weight

- Select the spring object in Outliner and enter **0** for the **End1 Weight** attribute.

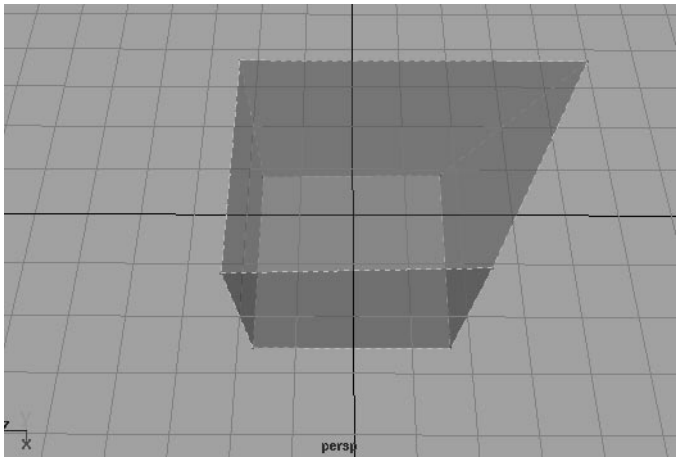
This will eliminate the force acting on the start of the spring. All of the force of the added **restLength** will be applied to the end of the spring.

End1 Weight and **End2 Weight** control at what percentage forces will act on either end of the spring.

1 (default) = 100%

0 = 0%.

- Playback



Spring End1 weight set to 0

Exercises: Zeppelin

The file *zeppelinStart.mb* contains a dirigible, a simpleSphere object and the ground.

- Use the *simpleSphere* object as a wrap deformer for the *AirShip*.
- Turn *simpleSphere* into a soft body and spring it up.
- Make the plane a passive rigid body
- Set the ground as a collision object to the soft body particles.
- Apply gravity to the soft body and watch it crash.

SUMMARY

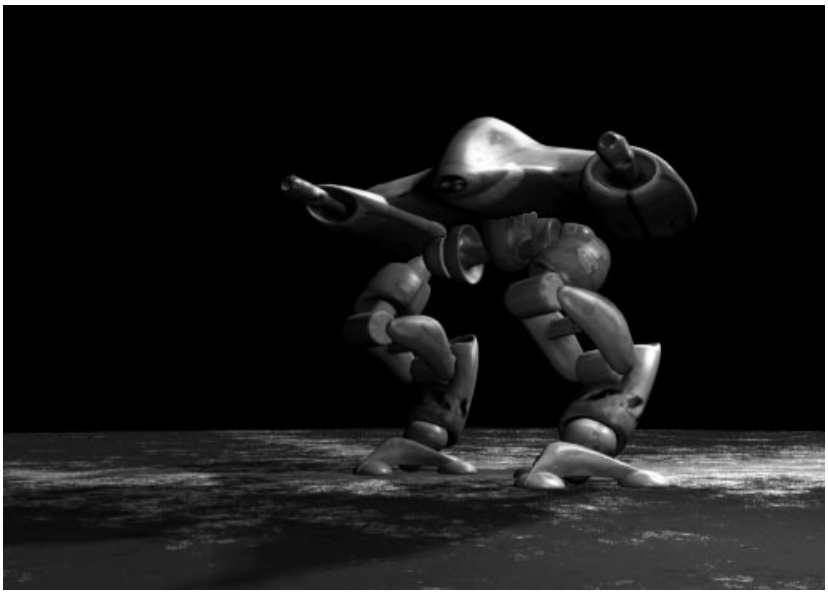
Soft bodies allow a direct manipulation of surface geometry with dynamic forces. There are several applications for this feature including modeling collision modeling, and fluid effects. The main limitation to using soft bodies is that the particles do not collide with each other or obey interpenetration checking of the surface. This is where cloth takes over for dynamically simulating fabrics and materials that must react to themselves.

Chapter 14

14 Compositing Dynamics

This lesson covers rendering strategies for compositing dynamic effects. In this lesson:

- Layers
- Rendering options for compositing
- UseBackGround shader
- Shadow passes
- Geometry masking



Compositing Is a Time Saver

Regardless of the job at hand, if you can find a way to do it faster, smarter and with the mind set that lifes little curves will be thrown at you, DO IT!

One such method of optimizing your production time is to always plan and prepare your project around compositing. Compositing has been an integral part of image creation since artists first began committing their images to some form of media. Film making pioneers were quick to grasp the power of combining image elements into a seamless sandwich of layers to produce final projected images that would not be possible otherwise.

CGI is no different and, in fact, has been a prime beneficiary of this process.

Separating your image elements into distinct rendered passes has the following advantages:

- Faster render times
- Flexibility of version injection and artistic control
- Faster and more precise color matching
- Sweetening processes and post process effects
- Lower resolution demands for inserted elements
- Hardware particle rendering

Larger and more elaborate concepts are made possible by compositing. For this reason all major studios have centered their image creation pipeline around the compositing process. The compositing station is the hub that all elements are fed into.

Maya provides tools to aid the process of image creation for compositing:

- Layer Management
- Shading options for Matte Opacity, BlackHole, UseBackground
- Geometry Masking
- Z-depth and Alpha / Matte rendering
- Object Visibility
- Shadow Pass Rendering / Shadow Catching

Example: Bye Bye Gunbot

This example utilizes much of what you have learned and applies it to a scene that is in the process of layer creation and manipulation for the compositing process.

In this shot you have a mechanical biped that is to lose its upper torso in a most violent manner.

Although the scene file has been broken down into layers for easy display management, some of the actual rendered layers are derived from set management / rendering flags as well.

Layer 1: GunbotPieces

These are the upper torso pieces that are shattering and flying away, they are rigid bodies that have been animated with various dynamic fields.

Layer2: ground

This is the ground plane. It is a passive rigid body.

Layer3: Legs

This is the Gunbot without the upper torso.

Layer4: BlastWave

A sphere that is animated to mimic the initial shockwave and act as a guide for pyro timing.

Layer5: GunBotBody

This is the upper torso intact. We will swap this object for the pieces objects at go time.

Layer6: GutsLeaders

This is a particle layer that consists of leading particle emitters. These are hardware rendered streaks.

Layer7: GutsSmoke

This layer is made up of the particles emitted from the GutsLeaders, They are software rendered clouds.

Layer8: ChunkSparks

This is a layer of particles that have been emitted from the surface of the gunbotPieces.

Layer9: MattePieces

Layer10: MatteLegs

Layer11: MatteGround

These layers are of duplicated geometry that is parented to the gunbot and ground surfaces. These objects have usebackground shaders applied to them for use in creating mattes and holdouts of render passes.

Layers Breakdown

Now let's talk about the individual rendering passes and how they fit together. Typical effects shots are likely much more complex. The emphasis here is on how the rendering passes were conceived not on the steps involved in actually compositing.

Layer1 movie: gunbotPieces

The gunbotPieces are a layer of rigidbody NURBs surfaces. They were derived from detached surfaces obtained from the original gunbot body. Also the gunpods are included and are children of standIn spheres that make up the actual Active Rigid bodies.

These pieces were animated using a radial field and gravity. The radial field attributes as well as the rigid bodies have keyframes on several attributes to add control to the accelerations.

These objects were rendered in software by themselves with the ground visibility turned off via its layer. The ground still acts as a passive rigid body collision for the gunbotPieces when hidden.

These objects are also rendered with z-depth to aid in compositing. But this is a technique that you do not want to rely on as it can lead to accuracy problems for objects that overlap or are very close.

Layer2 movie: ground

This is simply the ground plane software rendered. The ground shadow pass is derived from this layer by applying the useBackground to the ground object then rendering with primary visibility turned off on all of the gunbot objects: (gunbotPieces and legs).

Layer3 movie: blastwave

This layer is an animated NURBs sphere. It is software rendered with an x-ray or ghost shader applied. This shader creates the soft edge effect by using the Facing Ratio of the SamplerInfo utility node to drive the

transparency of the material. This layer is useful for first timing the rate of explosion and helps to coordinate all the elements of the explosion.

Layer4 movie: Guts

These particles are hardware rendered to simulate chunks of molten debris flying out at high velocity. They are rendered as points and with the geometry of the gunbot visible but with the geometry mask option enabled under the hardware render attributes.

Layer5 movie: gutsSmoke

These particles are emitted from the guts particles and have only a slight amount of Inherit Velocity. They also have their own gravity which is very slight. The intended effect is that they are trailing smoke. These are rendered in software. The geometry of the gunbot and ground are masked by using the useBackGround shader with matte opacity set to "blackHole".

Layer6 movie: Sparks

This layer of particles are surface emitted from the rigid body pieces. They have had the pixie dust treatment done to them so that they sparkle and flash. They collide with the ground and the gunbot. They have a collision event that emits other sparks at the point of collision. They are hardware rendered.

Layer7 movie: FireBall

This is a pyrotechnic film sequence. Because the plates do not have z-depth information a move3d event is applied to scale, position and add z-depth information. The TimeWarp event has also been used to correct the timing and duration.

Layer8 movie: Shadow Pass

This sequence was created by using the "useBackground" shader on the gunbot and the ground. With their Primary Visibility left on the geometry acts as shadow catchers. This information is only visible in the matte channel. To see your matte information when test rendering select Display ->Mask Plane. Composer will allow us to manipulate this information further by using it as a mask input channel to a "Brightness" event for example, thus recreating the shadows as darker areas on the Brightness events applied images.

Compositing

As the layers or passes are rendered they are tested together in your compositing application. Maya Composer and Maya Fusion are designed to anticipate the interchange between 3D and 2D.

Images that have been rendered in Maya are brought into the Compositing application as references only. As subsequent improvements or versions are created they can directly replace referenced images in the compositing "script".

Images can undergo drastic manipulation during compositing with much less rendering time. Lighting effects and manipulation of shadow color

and intensity as well as softness are a prime example. Rendering shadows separately and with coarse resolution with the intention of softening during composition can be a huge time and effort saver in itself.

Color correction and contrast balance as well as edge contouring (pseudo anti aliasing), film grain, camera shake, and lighting effects such as glow and lens flare are some of the popular effects achieved during compositing. All with very fast rendering updates.

Compositing is also where elements created in other applications are brought together. Maya Composer and Maya Fusion accept all the most popular film and video formats. These packages are also a great front end to bringing in external plates or video source footage for interaction as rotoscoping or image plane / texture elements in Maya.

SUMMARY

Compositing Advantages

Combining Software and Hardware rendered Particles

Object and Material options for visibility and lighting

Duplicating and parenting of child matte objects

A Expression Appendix

This appendix provides an in depth discussion of the expressions and steps used to build the examples contained in the file *expressionExamples.mb***.** You can use this appendix to walk you through how each example in that file was built and also get a description of how the expressions work.

This section also discusses the *order of evaluation* for the various elements in Maya's dynamics.

MOVING PARTICLES WITH EXPRESSIONS

Up to this point, we have been dealing primarily with using expressions to control rendering attributes such as color or opacity. We can apply similar methods to position, velocity or acceleration to dynamically control the motion of the particles.

Write a simple expression to control position

1 Create a particle in the scene

- Use the particle tool to create a single particle near the origin.
- Set **Render Type** to **Sphere.**

2 Add a runtime expression

- Add the following to runtime expression to **position**:

```
position = <<0, time, 0 >>;
```

3 Test the results

- Set the frame range to start at **1** and end at **300**
- Rewind; then playback

  The particle moves up in Y as time increases. As the animation plays back, *time* is a constantly changing value determined using the following relationship :

```
Time = Current Frame Number / Frames Per Second
```

Expressions for creating random motion

- Create a cloud of **100** particles.

- Try each of the following **by itself** in the runtime expression to see the interesting effects they produce:

```
velocity = dnoise (position);
acceleration = dnoise (position);
velocity = sphrand(10);
acceleration = sphrand(10);
position = position + dnoise(position);
```

An acceleration rule that uses variables and magnitude

1 Create a new scene file with a cloud of particles in it

- Select **File** → **New.**
- Create a cloud of **50** particles using the Particle Tool.
- Set **Render type** to **Sphere.**
- Set **Radius** to **0.3**

2 Add a runtime expression to acceleration

- Enter the following in the runtime expression for **acceleration**

```
int $frequency = 65;
float $distance = mag (position);
int $limit = 3;


if ($distance > $limit)
acceleration = acceleration - (position *
$frequency);
```

- Click **Create** in the Expression Editor.

3 Test the results

- Playback the animation.

  The particles move in a swarming pattern. Watch one particle to see what it is doing. It is swinging between a range in 3D space defined by $limit. When the **magnitude** of the position is greater than that limit, the expression begins substracting acceleration from the particle which increases its acceleration in the opposite direction.

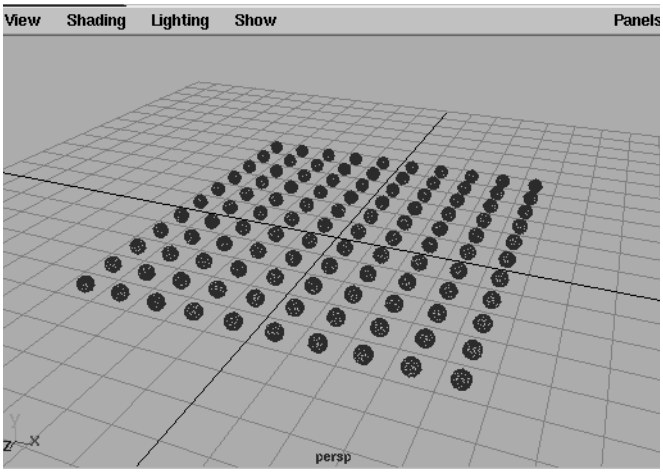
  If this is unclear to you, try the same expression on a single particle instead of a cloud. Also, try changing the values used for frequency and limit.

  You can also try this same effect with emitted particles.

A position rule that uses noise and a custom attribute

1 Create a new scene with a grid of particles

- Use the Particle Tool to create a grid of particles that has some particles in all 4 quadrants of the grid
- Set **Render Type** to **Sphere**
- Set **Radius** to **0.3**



A grid of particles in all four quadrants

2 Make a custom vector attribute to store original position

Adding a custom vector attribute will give us a place to store the original position of each particle.

- Select *particleShape1*.
- **Modify** → **Add Attribute...**

  Attribute Name to **origPos**

  Data Type to **Vector**

  Attribute Type to **Array**

  Add Initial State Attribute to **ON**
- Press **OK.**

  An *origPos* field is added to the Per Particle (Array) Attributes section of the Attribute Editor.

  Since the **Attribute Type** was set to **Array**, you just made your own custom per particle attribute.

3 Add a creation expression for origPos

- Add the following creation expression to *origPos* to store the postion of each particle at the intial frame.

```
origPos = position;
```

4 Add a runtime expression to position

- Click the **Runtime** radio button in the Expression Editor.
- Enter the following runtime expression to control **position** :

```
position = origPos + <<0, 0.8 * noise (origPos *
3+time*<<0,1,2>>) ,0>>
```

5 Test the results

- Rewind; then playback

  Each particle moves in a wave-like fashion up and down along **only** the Y axis. The above expression is just adding a vector to origPos. The Y component of that vector is a statement that generates a random stream. **0.8** controls the amplitude of that stream, **3** controls the frequency, **<<0,1,2>>** controls the direction of the phase.

  The `noise` function produces a smoother random number stream than the previously discussed `rand` function.

Note: A plotted graph for of the noise function is available in the **functions** section of the online MEL documentation.

---

6 Save the file

- Save the file as **waveExpression.ma**

Change Color based on Position

- Create an **Omni Directional** emitter with the default options.
- Add an **rgbPP** attribute.
- Add the following runtime expression to position

```
vector $pos = position;
if ($pos.y >=0)
rgbPP = <<1,0,0>>;
else
rgbPP = <<0,0,1>>;
```

- Rewind and playback

  This expression stores the position for each particle in a vector variable called **$pos**. The **if** statement checks the Y **component** of that vector to see if it is above or below the Y axis. If the particle is above, it is red, otherwise it is blue.

  You can try the same idea with acceleration or velocity instead of position.

Tip: The individual elements of a vector (i.e. <<x,y,z>>) are called components .

Emitter examples

Although emitters are closely related to particles, they **do not** use creation and runtime expressions. Below are two examples that show you some ideas you can build upon when working with emitters.

Varying the emission rate

The following example is good for obtaining a randomized emission rate that can be used to simulate an eruption, geiser, or puffing smoke effect :

1 Start with an empty scene file

- Select **File** → **New.**

2 Create a directional emitter emitting particles in Y

- **Speed** to **5**
- **Direction** to **0, 1, 0** for **x**, **y**, and **z** respectively
- **Render Type** to **Clouds**
- **Radius** to **0.2**

3 Add gravity to the particles

- Select the particles then **Fields**→**Create Gravity**

4 Add an expression to control rate

- Select **emitter1**
- Open the Expression Editor and enter the following expression

```
emitter1.rate = 1000*noise (time*1000);
```

This expression uses **noise** as opposed to **dnoise** since rate is a **scalar** quantity. We would only use **dnoise** if we were working with a **vector** quantity such as position or color.

We multiply noise and time by 1000 to increase the amplitude and frequency of the noise values since they are far too small without these multipliers.

PARTICLE ID

Just as each building on a street has its own address number, each particle in a particle object has its own unique numerical identity called the **particleId**. ParticleId is an integer value ranging from **0** to **count - 1**.

The particleId attribute makes it easier to control attributes of specific particles independently of other particles within the same particle object. This is especially useful for adding variation to attributes of a particle object.

Control color based on particleId

1 Create a new scene file

- **File**→**New**.

2 Create an Omni emitter with default values

- Select **Particles** → **Create Emitter.**
- Set **Emitter Type** to **Omni.**

3 Add an rgbPP attribute to particleShape1

4 Add a runtime expression to rgbPP

- Add the following to the runtime expression for *particleShape1*

```
if (particleId==10)
rgbPP = <<0,1,0>>;
```

- Press **Edit** in the Expression Editor.
- Rewind; then play the animation.

  The first particle in a particle object is always particleId **0**. Therefore, the 9th particle emitted into particleShape2 is particleId **10** and is colored green due to the **if** conditional statement in the expression.

  Below is another particleId example you can add as runtime or creation expression for rgbPP to produce some interesting results

```
if (particleId % 10 == 0)
rgbPP = sphrand(1);
```

  The **%** symbol stands for the **modulus operation** which is the remainder produced when two numbers are divided. The above expression divides the particleId by 10, if the **remainder** of that division is 0 then the the **sphrand** function picks a random vector value between <<0,0,0>> and <<1,1,1>>. In other words, every 10th particle will get a random color assigned.

ORDER OF EVALUATION

The following is a breakdown of the order in which Maya evaluates the various dynamic elements in a simulation.

- First, the **acceleration** is cleared at the beginning of each frame or evaluation.
- **Particle expressions** are evauated secondly. The expressions can get, set, or add to the current values of the particle's attributes.
- Next, the **forces** are computed. Forces include **fields, springs,** and **goals**. These forces are added to whatever is currently in the acceleration which includes whatever a particle expression may have put there.
- The **velocity** is computed from the acceleration. This also just adds to whatever value is currently in the velocity which may have previously been set in an expression.
- Finally, the **positions** are computed from the velocity. Just as with acceleration and velocity, position is added to whatever is currently stored in position from expressions or forces computed above.

The expressions do not override the dynamics. The dynamics happen after the expressions are evaluated, and their results are added together. It is possible to have expressions calculated before dynamics on a per object basis by disabling the **Expressions After Dynamics** checkbox of the particleShape object.

B Emit Appendix

This appendix provides detailed descriptions of the expressions used in the Emit function lesson.

Fireworks1 Expression :

Below is the runtime expression used for *fireworksShape1*. The short command flags have been replaced with the long flagnames for additional clarity. A detailed description is provided after the expression.

```
if ($vel.y < 0)
{ //opening bracket for the if statement
fireworksShape1.lifespanPP = 0;
float $antiGrav = launcher.antiGrav;
int $upperCount = launcher.showerUpper;
int $lowerCount = launcher.showerLower;
int $upperLife = launcher.streamUpper;
int $lowerLife = launcher.streamLower;
int $numPars = rand($lowerCount, $upperCount);

string $emitCmd = "emit -object fireworksShape2 ";

for ($i=1; $i<=$numPars; $i++)
  { //opening bracket for the for loop
    $emitCmd += "-position " + $pos + " ";
    vector $vrand = sphrand(10);
    $vrand = <<$vrand.x, $vrand.y + $antiGrav,
$vrand.z>>;
    $emitCmd += "-attribute velocity ";
    $emitCmd += "-vectorValue " + $vrand + " ";
    float $lsrand = rand($lowerLife, $upperLife);
    $emitCmd += "-attribute lifespanPP ";
    $emitCmd += "-floatValue " + $lsrand + " ";
```

```
    } //closing bracket for the for loop

    eval($emitCmd);
  } //closing bracket for the if statement
```

Step by step explanation :

```
float $antiGrav = launcher.antiGrav;
```

- Store the value for **anitGrav** into a float (decimal) variable called **$antiGrav**. **AntiGrav** is one of the **custom** attributes previously added to **launcher**.

  The $antiGrav variable will be used later in this expression to add or remove velocity in the Y direction as particles fall. This provides a way to add to or counteract the effect of gravity.

```
int $upperCount = launcher.showerUpper;
int $lowerCount = launcher.showerLower;
```

- **showerLower** and **showerUpper** are two of the **custom** attributes previously added to **launcher**.
- These attributes define a range (lower and upper bound) out of which a random number will be picked later in the expression.
- That random number will be used to control the **number of particles** emitted into **fireworks2**.

```
int $upperLife = launcher.streamUpper;
int $lowerLife = launcher.streamLower;
```

- **streamUpper** and **streamLower** are two more of the **custom** attributes previously added to **launcher**.
- These attributes define a range (lower and upper bound) out of which a random number will be chosen later in the expression.
- That random number will be used to control the **lifespan** of particles emitted into **fireworsk2.**

```
int $numPars = rand($lowerCount, $upperCount);
```

- Choose a random integer number from within the range of values defined by **$lowerCount** and **$upperCount**.
- Assign that random value to **$numPars**.
- **$numPars** will be used later in this expression to control the **number of times** the commands within a loop will be executed.

```
string $emitCmd = "emit -object fireworksShape2 ";
```

- The remaining portion of the expression is designed to piece together the emit function and then execute it once it has been fully assembled.

- Each particle created in **fireworksShape2** will be the result of using the same basic syntax framework for the emit function. The only difference will be substituting in different attribute values (position, velocity, etc) for each particle.
- **$emitCmd** stores the emit command while it is being constructed in the expression. The bold text above is the first piece of the emit function. The remaining elements will be constructed using the following looping structure.

```
for ($i=1; $i<=$numPars; $i++)
```

- This is a looping structure (for loop) that will exectute the commands following it that are enclosed by curly brackets.
- The number of times those commands are executed is controlled by the random value assigned to **$numParts**.
- The basic syntax of a **for loop** is :

```
for (startValue; endValue; increment).
{
statements;
}
```

In the expresion's loop, **$i** is the **startValue** and represents how many times you've cycled through the loop.

The first time through the loop **$i** is the **startValue** (1).

Then, **$i** is incremented by **1** (this is what **i++** in the increment does).

Therefore, the second time through the loop **$i** =**2**.

As long as the condition defined by **endValue** (**$i<=$numPars**) is true, **$i** will be incremented and the loop will continue.

When the **endValue** condition is false, the loop is exited and the next line in the expression is evaluated.

```
$emitCmd += "-position " + $pos + " ";
    vector $vrand = sphrand(10);
    $vrand = <<$vrand.x, $vrand.y + $antiGrav, $vrand.z>>;
    $emitCmd += "-attribute velocity ";
    $emitCmd += "-vectorValue " + $vrand + " ";
    float $lsrand = rand($lowerLife, $upperLife);
    $emitCmd += "-attribute lifespanPP ";
    $emitCmd += "-floatValue " + $lsrand + " ";
```

- In this case, the sytnax convention **+=** means "take what is currently stored in **$emitCmd** and append what is on the right side of the symbol to the end of **$emitCmd**."

- In the first excercise of the emit function lesson, you constructed 3 emit statements to add three individual particles to a particle object. The contents of the for loop listed here does the same thing many times. Each line above is executed once for each iteration of the for loop.
- $vrand uses the **sphrand** function to select a random vector value between <<0,0,0>> and <<10,10,10>>. This provides a random value to use for velocity.
- The expression constantly appends to **$emitCmd**. Each line is setting a different attribute for the emit command.

```
eval($emitCmd);
```

- The **eval** command is a **MEL** command that functions much like the **=** button on a calculator. However, in this case we are "inputting" letters and numbers instead of only numbers.
- The for loop was like entering all the values in the calculator to construct the **$emitCmd** variable.
- After the loop has been finished, the expression evaluates the contents of **$emitCmd** to actually place the particle in the correct locations and assign them the correct attribute values.

Note: There is more information and examples of the usage for the **emit** function, **sphrand**, **rand**, **eval**, and **for loops** in the online **MEL** documentation.
