

Creating Models for Simulations

MultiGen Creator
Version 2.5 for Windows and IRIX
August 2001

USE AND DISCLOSURE OF DATA

© Computer Associates, 2001. All rights reserved. MultiGen-Paradigm, Inc., a Computer Associates Company, is the owner of all intellectual property rights, including but not limited to, copyrights in and to this document and its contents. Use of this document is subject to the terms of the MultiGen-Paradigm Software License Agreement included with this product. This document may not be reproduced or distributed in any form, in whole or in part, without the express written permission of MultiGen-Paradigm, Inc.



***Creating Models for Simulations, Version 2.5 for Windows and IRIX
August 2001***

MultiGen-Paradigm, Inc. (MultiGen-Paradigm), a Computer Associates Company, PROVIDES THIS MATERIAL AS IS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MultiGen-Paradigm, Inc., a Computer Associates Company, may make improvements and changes to the product described in this manual at any time without notice. MultiGen-Paradigm assumes no responsibility for the use of the product or this manual except as expressly set forth in the applicable MultiGen-Paradigm agreement or agreements and subject to terms and conditions set forth therein and applicable MultiGen-Paradigm policies and procedures. This manual may contain technical inaccuracies or typographical errors. Periodic changes are made to the information contained herein: These changes will be incorporated in new editions of the manual.

Copyright ©2001 by Computer Associates. World rights reserved.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, digital, or other record, without the prior written permission of MultiGen-Paradigm, Inc.

Use, duplication, or disclosure by the government is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause and DFARS 52.227-7013 and in similar clauses in the FAR and NASA FAR Supplement.

MultiGen, OpenFlight, and Flight Format are registered trademarks of MultiGen-Paradigm, Inc., a Computer Associates Company. MultiGen II, MultiGen Creator, and Vega are trademarks of MultiGen-Paradigm, Inc., a Computer Associates Company.

Adobe Photoshop is a trademark of Adobe System, Inc., which may be registered in certain jurisdictions.

MS, MS-DOS, Windows and Windows NT are registered trademarks of Microsoft Corporation in the United States and/or other countries.

OpenGL and IRIX are registered trademarks, and IRIS Performer and SGI are trademarks of Silicon Graphics, Inc.

All other trademarks herein belong to their respective owners.

Printed in the U.S.A.

Part Number: DBMC03 08/01

Contents

Chapter 1 Background

What is the Difference Between a Realtime Application and an Animation? ..	1-2
What are 3D Graphics?	1-3
The Basic Realtime Application Process	1-5

Chapter 2 Structuring a Database

Anatomy of a Database	2-2
Basic Node Types	2-3
Database Node Organization	2-4
Selecting Modes	2-5
Issues to Consider	2-7

Chapter 3 What Happens at Runtime

The Rendering Process	3-2
Understanding Application, Cull, and Draw	3-3
Balancing Culling and Drawing	3-4
Defining a Viewing Volume	3-5
Depth Complexity	3-6
State Changes	3-7
Moving Nodes in the Hierarchy	3-7

Chapter 4 Exploring Modeling Techniques

Using the Tracking Plane	4-1
Positioning the Tracking Plane	4-2
Placing Items Onto Other Geometry	4-3
Slicing Geometry Along a Plane	4-4
Modifying Geometry	4-5
Mapping Textures to Geometry	4-7
Creating Planar and Nonplanar Faces	4-7
Modeling with Polygons	4-9
Applying Textures	4-11
Texture Mapping	4-11
The Put Texture Tools	4-12
The Surface Project Texture Tool	4-15

- The Spherical Project Texture Tool4-15
- The Radial Project Texture Tool4-16
- The Environment Map Texture Tool4-17
- Other Texture Techniques4-18
- Reducing Texture Memory4-23
 - Reducing Internal Data Formats4-24
 - Applying Subtextures4-25
 - A Note on Graphics Card Support4-26
- Lighting and Shading Your Database4-27
 - Adding Light Sources4-29
 - Changing Light Intensity4-36
 - Shading an Object4-38
 - Lighting and Materials4-41
 - Creating Shadows and Other Effects4-42
- Adding Light Points4-46
 - Light Point Parameters4-46
 - Light Point Lobes4-51
 - Defining Light Points for LODs4-52
- Adding Movement4-52
 - Creating a DOF Local Coordinate System4-53
 - Defining Movement4-54
 - Checking DOF Movement4-55
- Adding Sound4-55
 - Loading Sounds4-56
 - Creating a Sound Node4-57

Chapter 5 Exploring Methods to Simplify Modeling

- Defining a Local Coordinate System5-1
 - Defining a Local Coordinate System With a Transformation5-2
 - Moving an Object5-3
- Creating Construction Edges and Curves5-3
 - Construction Edges5-3
 - Construction Curves5-4
- Using Background Images5-8
- Creating Billboards5-9
- Using External References5-10
- Using Instances5-11
- Applying a Transformation Edit5-12

Chapter 6 Optimizing for Performance

Structuring the Hierarchy for Efficiency6-2
 The Cull Process6-2
 Draw Order6-7
 Using Bounding Volumes6-15
 Reducing Polygons6-17
 Using Levels of Detail6-17
 Replacing Polygons with Texture6-21
 Removing Unnecessary Polygons6-22
 Removing Back Faces of Polygons6-22
 Moving Clipping Planes6-23

Index Index-1

1 **Background**

MultiGen Creator is a highly-specialized tool that helps modelers produce efficient three-dimensional (3D) models and terrain for interactive realtime applications. Interactive applications are diverse in nature, ranging from flight and vehicle training simulations for military personnel to visual demonstrations of construction projects for architects.

More than a basic modeling tool, Creator is a design tool for building low-polygon models that simplify and reduce programming requirements for the realtime application. Creator provides a user interface for constructing models, terrain, and scenes in a hierarchical visual database that conforms to the OpenFlight standard file format (.flt). The OpenFlight file becomes part of a realtime application after it is imported into runtime software such as MultiGen-Paradigm's Vega.

Before you begin to model with Creator, it is useful to understand the concepts behind modeling for realtime that are described in this chapter, including 3D graphics, the differences between realtime and animation, and the process for creating a realtime simulation with MultiGen-Paradigm's tools. The other chapters in this book provide explanations of modeling techniques and tools to help you model in Creator and achieve your desired results in the realtime application.

What is the Difference Between a Realtime Application and an Animation?

Animations are used for films, images for print, and preprogrammed demonstrations. Realtime applications are used in situations where responding to user input is part of the simulation, for example, during flight training, video games, and interactive architectural demonstrations. Both realtime and animation applications simulate real and imaginary worlds with highly detailed models, produce smooth continuous movement, and render at a certain number of frames per second for seamless presentation. The main differences?

- **Realtime application** frames are rendered in *real time*, which means the frames are continuously recalculated and rendered as the user changes direction and chooses where to move through the scene to view; **Animation** frames are *pre-rendered*, which means the animator sets the order of the frames and chooses the parts of the scene to view. Each frame can take hours to render.
- **Realtime applications** are highly interactive, and the user controls the movement of objects within the scene; **animations** do not allow for human interaction, and the user is a passive participant.
- The emphases of **realtime applications** are interactivity and purpose. Models in realtime applications usually have fewer details than models in animations to increase the rendering speed and shorten the *latency period*, which is the time delay from user input until the application makes an appropriate response. To achieve realistic realtime simulations, the latency period must be too short for the user to perceive.

The emphases of **animations** are non-interactive aesthetics and visual effects. Models in animations usually have more details: because the frames are pre-rendered, the effect on drawing speed can be pre-determined.

- **Realtime applications** display at various frame rates, which range from 15-60 frames per second, depending on application goals and screen complexity; **animations** usually display at 24 frames per second for every pre-rendered sequence of images, which is a predictable number.

What are 3D Graphics?

A two-dimensional (2D) computer-generated image has only horizontal and vertical coordinates on the x and y -axes. A 3D image has x and y coordinates as well as a z coordinate on the z -axis that defines an extra dimension (depth). When shading and textures are applied to a 3D object, the object appears to be much more realistic than an object drawn in two dimensions. You can move through and around 3D models and images in 3D space, which provides the experience of traveling through a virtual world.

The 3D models that you create are *tessellated* into *simple, convex* polygons. Tessellation is the process of converting an object into a collection of polygons. A polygon is simple if edges intersect only at vertices and exactly two edges meet at any vertex, such as a square. A polygon is convex if there are no indentations and a line joining any two points in the interior of the polygon also lies in the interior.

The polygon's simple and convex properties make it acceptable for OpenGL, a software interface for graphics hardware that Creator uses, to display the polygon. Other types of polygons that vary in shape can be very complicated to draw, and OpenGL might not produce results that you expect.



Valid Polygons

These polygons are both *simple* and *convex*. Their edges intersect only at vertices, and lines joining any two points in the interior also lie in the interior.

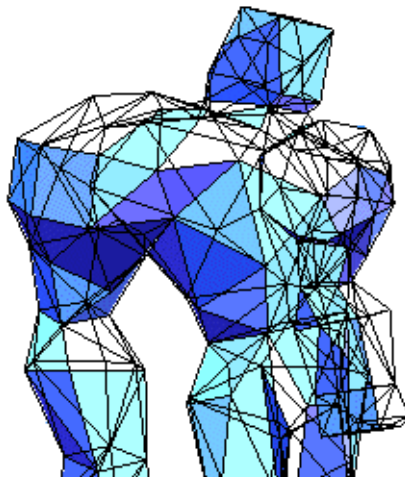
Invalid Polygons

The first polygon is *complex* because it has crossing edges. All three polygons are *concave* because lines can join points in the interior and also lie in the exterior of the polygon.

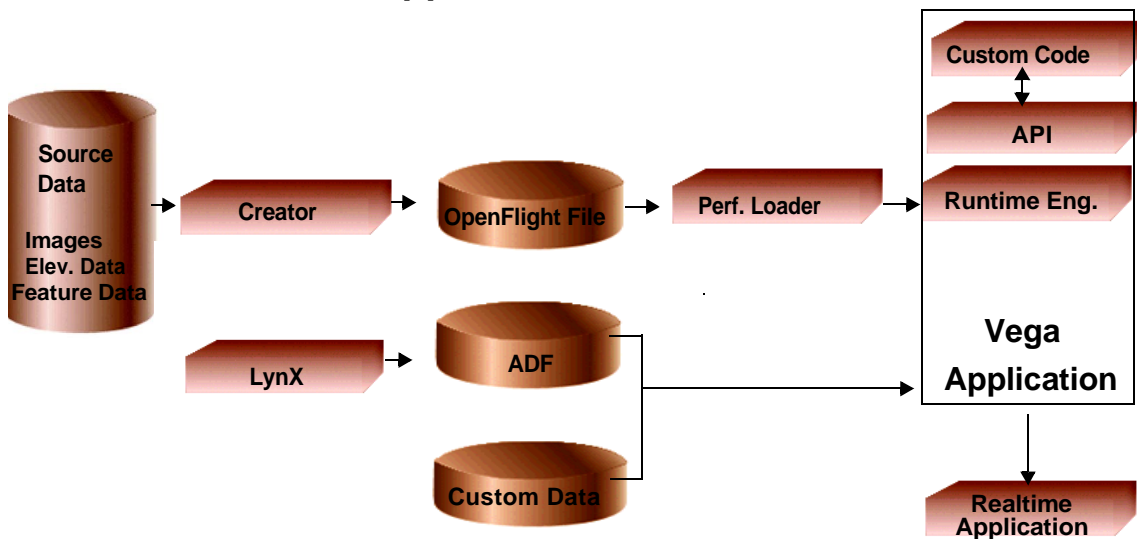
The most basic type of polygon that meets OpenGL requirements is the triangle. Creator divides concave polygons, such as the second polygon shown in the illustration of invalid polygons, into triangles and other regular shapes for OpenGL to easily render. Both Creator and OpenGL can have difficulties displaying the first, complex polygon correctly. As you model, it is a good idea to use mostly simple polygons for best results.

Triangles are optimal for Creator because:

- Triangles are simple, convex polygons.
- The triangle's vertices are always *coplanar*. Coplanar vertices lie in the same plane. OpenGL draws coplanar polygons correctly when they are projected from 3D space onto a 2D display. OpenGL also draws them correctly when they are rotated in 3D space. Polygons with noncoplanar vertices (*nonplanar*) can become complex and concave when rotated, and OpenGL might not render them correctly.



The Basic Realtime Application Process



MultiGen-Paradigm's software products, Creator and Vega, are primary tools used in the process of creating a 3D realtime application. This illustration shows a basic realtime application development process on an SGI computer system:

- Using Creator terrain tools, you can convert source data, such as satellite images and digital elevation data, to the Creator file format and import the converted files into Creator to create terrain skin. Source data also includes feature data that you can convert and import into Creator for adding culture to your terrain, such as roads and buildings.
- Using Creator modeling tools, you can hand-create 3D models. You can apply attributes such as color, materials, and textures to add realism to your terrain, culture, and models. All of these elements, terrain, features, models, and attributes, comprise the Creator visual database that the OpenGL application programming interface (API) supports. The visual database is saved in MultiGen-Paradigm's OpenFlight (.flt) file format, which has become the standard file format for most realtime systems.

- You can use the Vega development environment to create the realtime application. The Vega LynX utility provides a graphical user interface for setting up an Application Definition File (ADF) for the realtime application. The ADF describes the OpenFlight files used in the realtime application, moving models and their paths, special effects such as explosions and environmental effects, and other functionality. In addition to the ADF, you can write custom data to support the needs of your simulation.
- You will most likely write custom code within the Vega development environment and software library to produce a stand-alone realtime application. The Vega environment includes the API, custom code, and runtime engine, such as the IRIS Performer-based Perfly for SGI Irix.
- The IRIS Performer loader loads the OpenFlight file created in Creator into Vega on an image generator (IG), which is the graphical hardware that draws the scene. Image generators can be Reality Engines, InfiniteRealities, standard PCs, or gaming consoles, to name a few. The ADF and any custom data are also loaded into Vega.
- After the realtime application is compiled, it is ready to run. Together, the realtime application and computer platform are often referred to as the *runtime system*.

2 *Structuring a Database*

Models must be defined with geometry, hierarchy, and attributes for realtime applications. Creator database files use the OpenFlight scene-description format for modeling that requires these components. You have control and flexibility with adding these components in the Creator database.

The database hierarchy that the OpenFlight format uses has two main purposes: it organizes geometry into *nodes* that you can easily edit and move, and it provides a tree structure that the runtime system can process. A node is the fundamental element or building block for constructing the database hierarchy. A node is often referred to as a bead or geode in other software products. You can design your database hierarchy to help your simulation run smoothly, accurately, and quickly if you understand:

- The basic components (geometry, hierarchy, attributes) in a database
- The types of nodes, how to create and edit them, and how to organize them in a database hierarchy
- Issues to consider for structuring the database to meet your runtime system's requirements

Basic Node Types

The basic nodes that you use to build models in the database are group, object, face, and vertex. Group, object, and face nodes are placed underneath a default database header node in the Hierarchy view.

These are descriptions of the basic node types:

- A *database header node* is at the top of the database hierarchy. It is automatically created for each new file and is labeled **db** in the hierarchy. Information about the database, such as database units, creation and modification dates, is stored in the database header. You cannot delete or select this node.
- A *group node* represents a collection of object nodes or other logically-related nodes. Nodes that are logically organized into groups are easier to manipulate in the database. For example, instead of moving individual object nodes, you only need to move the parent group node to move all of them at once. Group nodes are color-coded red in the hierarchy, and default group node IDs always start with **g**.
- An *object node* represents a collection of face nodes. Faces that are logically organized into objects are easier to manipulate in the database. Object IDs are color-coded green in the hierarchy, and default object IDs always start with **o**.
- A *face node* represents a collection of ordered, coplanar vertices that describe a surface. A face node's color reflects the color assigned to the face in the Graphics view, and its default face node ID always starts with **p**.
- A *vertex node* represents a coordinate point in the database. Each coordinate point is defined by a unique set of three numbers. For example, the coordinates (0,0,0) define the center point in 3D space, which is also called the database origin. The unit of the database coordinate system, such as feet or meters, is determined by the **Database Units** attribute of the database header node. You create vertices during different edit functions, such as creating a face. Because models usually have a lot of vertices, vertex nodes are not visible in the database hierarchy to save space in the Hierarchy view.

You can add the following features to your database as separate nodes for special effects:

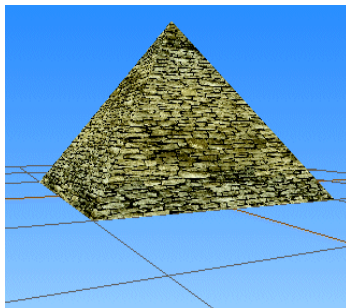
- *Degrees of Freedom (DOFs)* define a range of motion. The DOF node adds movement to the geometry below it in the database hierarchy.
- *Levels of Detail (LODs)* are versions of a model with varying degrees of complexity. The LOD node defines the range in which the LOD is visible.
- *Light sources* illuminate all or parts of your database. The light source node defines the position or direction of the light source and affects its descendants in the database hierarchy.
- *Sound files* add sound to your database. The sound node has a sound file assigned to it that plays from a location in the database that you choose.

These database features are discussed in more detail in this book.

Database Node Organization

An OpenFlight model is subdivided into several groupings in the database hierarchy so that the model's components are easy to edit. The entire model is organized under a group node, a section of the model is represented by an object node, a polygon in the object is a face node, and each face has edges and vertices that can be manipulated.

The first example of a pyramid model, which is in a database scene, illustrates the basic idea of how the individual nodes for a model are organized. The nodes are arranged in a hierarchical tree structure so that the runtime system can easily display the images as it reads them: top-to-bottom, left-to-right. The pyramid object is under a group node, and each of its faces is created as an individual face node.



The database header node

The group node represents the entire scene

The object node represents the pyramid

The face nodes represent the pyramid faces

Creator automatically arranges the pyramid's geometry in this structure as the model is created, but you can manually create nodes by using the **Create** tools, such as **Create Group** or **Create Object**, in the **Create** toolbox and place them at any level in the hierarchy.

Selecting Modes

You select a *modeling mode* for creating new nodes and for selecting geometry in the Graphics view. A mode reflects a level of organization in the database hierarchy.

When you select **Create** in the **Create** toolbox to create a new node, the node's type, such as group or object, is based on the current modeling mode. The current modeling mode appears in the **Mode** list in the Toolbar, as shown on the next page. You can use other **Create** tools, such as **Create Group** or **Create Object**, to create a new node in any mode.

You also select modes before you select the corresponding geometry in the Graphics view. For example, if you select a model in Vertex mode, you select its vertices; in Face mode, you select its faces that share the vertices; in Object mode, you select its objects that contain the faces and vertices; and so on.

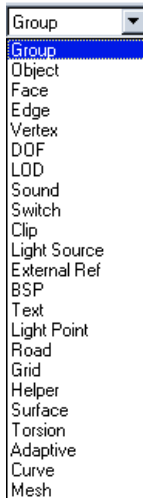
To change the mode in which an item is selected, you can select a new modeling mode in the **Mode** toolbox and choose **Select/Change Mode**. For example, if you select a face in Face mode, change to Vertex mode, and choose **Change Mode**, the vertices of the face will be selected. You can use this method instead of changing the mode and reselecting geometry in the Graphics view.









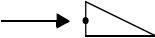


Each type of node in the hierarchy is supported by a mode, as shown in the following example. To edit a wing (an object node) of an airplane, you would first change to Object mode by choosing the **Object Mode** tool from the **Mode** toolbox or by selecting Object mode from the **Mode** list in the Toolbar. You could then select the wing's geometry in the Graphics view and make your edits. The parts of the airplane outside of that object node would not be selected, which makes it easier to edit or add detail to the airplane's wing. You could alternatively select the object node in the hierarchy in any mode to edit the wing.

Mode Toolbox With Modeling Mode Tools



Mode List



Modes		Database Nodes
		<input type="text" value="db"/>
 Group		<input type="text" value="spitfire"/>
 Object		<input type="text" value="wing"/>
 Face		<input type="text" value="p647"/>
 Edge		
 Vertex		

Issues to Consider

After you understand the basic idea of nodes and the database hierarchy, you can structure the nodes in your database in many different ways. How do you know what the best design is? You should structure your database according to your runtime needs. Proper database structure is important for creating models that meet your needs the first time they are drawn. Since geometry is drawn as the runtime system traverses the database hierarchy, you want to make sure pieces of your geometry appear in the correct order and that you see your desired effects.

You can use a combination of different structures in your database based on features and issues such as these:

- **Runtime performance** - A well-structured database hierarchy is important for smooth continuous movement in realtime. Chapter 6, *Optimizing for Performance* discusses how to structure the hierarchy for optimal runtime performance and speed.
- **Type of geometry, such as models or terrain** - The benefits of using one type of structure (logical structure) for a single model and a different type of structure (spatial structure) for a large-scale terrain database are discussed in Chapter 6, *Optimizing for Performance*.
- **DOFs** - Degrees of Freedom (DOFs) add movement to certain nodes. DOFs are discussed in Chapter 4, *Exploring Modeling Techniques*.
- **BSPs** - Binary Separating Planes (BSPs) ensure geometry is drawn in the correct order. BSPs are discussed in Chapter 6, *Optimizing for Performance*.
- **LODs** - Levels of Detail (LODs) are used for displaying certain parts of geometry at different distances from the eyepoint. LODs are discussed in Chapter 6, *Optimizing for Performance*.
- **Ease of correction and editing** - In large models with a lot of detail, you want to be able to quickly find the nodes to edit. Naming the nodes as well as arranging the nodes into large sections so you can select, isolate, and work on only those sections is important. You and other modelers can easily modify and maintain a well-structured database.

3 *What Happens at Runtime*

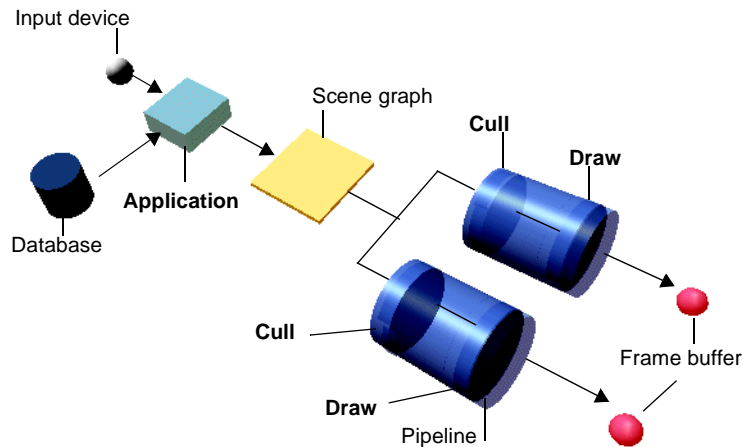
If you understand how the runtime system processes data, you can plan ahead of time how to design your database. The database you create needs to conform to your runtime system's operating limits. Since much of the runtime system's performance affects the quality of your realtime application, there are tasks that you can do in your database to optimize the performance. Careful planning of database design is key to a successful outcome. These runtime issues influence your database planning:

- The runtime system renders each frame of your database in three stages: *application*, *cull*, and *draw*. After the runtime system loads the database during the application stage, the runtime system must process the data in two additional stages to draw the viewable scene. During the cull stage, the runtime system traverses the database hierarchy to find the data in view. During the draw stage, the runtime system draws the data in view. For optimal performance, you want to design your database hierarchy so that the processing time for the cull and draw stages are balanced.
- Objects in the database scene are projected onto the screen using a *viewing volume*. You define a viewing volume in the runtime system to use a *perspective* or *orthographic* projection. Objects within the viewing volume have depth when they are projected using a perspective projection, in which objects closer to the eyepoint appear larger to the viewer. The database scene appears flat when objects are projected using an *orthographic projection*, in which object sizes are not dependent on their position.
- Polygons are converted to pixels before they are drawn on the screen. Pixels on overlapping polygons are drawn more than once, which increases the draw time and *depth complexity*. Depth complexity refers to the number of times each pixel is drawn per frame. You can use techniques to reduce the depth complexity and meet your runtime system's *pixel fill rate*, which is the number of pixels that your runtime system can draw per second.
- Draw time is increased with each *state change* in the database hierarchy. A state change occurs when a polygon with one set of attributes or states, such as color or texture, switches to another polygon with different states in the database hierarchy. You can rearrange nodes in the database hierarchy to reduce state changes.

The Rendering Process

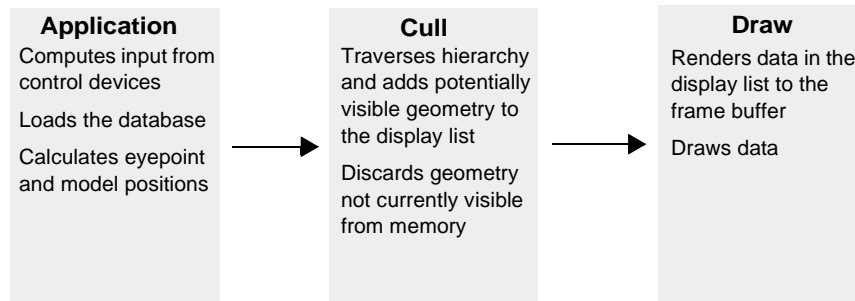
3D rendering software, such as IRIS Performer for SGI computer systems running the IRIX operating system, provides the rendering architecture for your realtime application. The architecture includes multithreaded (multiple process), parallel rendering pipelines for scene management and image generation that outputs to graphics pipelines.

The rendering process is split into three major stages: *application*, *cull*, and *draw*. The application stage loads the OpenFlight database and input from control devices, such as joysticks or console buttons, and constructs them into a *scene graph* (data structure to manage databases). The scene graph is sent to the software graphic pipeline, which consists of the cull and draw stages.



Understanding Application, Cull, and Draw

The runtime system processes each frame in your realtime application in three stages, *application*, *cull*, and *draw*, to maintain a consistent frame rate on the computer screen. These stages are outlined in this section.



Step 1 During the *application* stage, the runtime system reads input from control devices such as a joystick or console buttons, other simulation input, and positions of the eyepoint. Models are computed, interactivity of any networked simulators is conducted, and information is sent to instruments and controls to compute the final scene.

Step 2 During the *cull* stage, the runtime system traverses the database hierarchy and finds the parts of the database that are potentially visible based on their *bounding volumes*, such as a box, sphere, or cylinder, that enclose geometry for determining the parts that are in view. Models are culled when the models are out of view to improve runtime performance. Bounding volumes are discussed more in detail in “Using Bounding Volumes” on page 6-15.

As part of the cull stage, the runtime system checks if the bounding volumes intersect the *viewing volume*, which is the portion of the database that is visible in the Graphics view, and discards (or culls) the bounding volumes with geometry that do not intersect. The data that survives the cull stage is saved in a display list and stored in memory for the draw stage.

Step 3 During the *draw* stage, the runtime system renders the data, such as polygons and texture, that is in the display list into the frame buffer to draw. The display list is only used for one frame and is discarded before the next application, cull, and draw cycle.

Balancing Culling and Drawing

The runtime system should spend about the same amount of time culling the data to be viewed as it does to draw images. The realtime application might be slow or miss the target frame rate if the cull and draw stages are unbalanced. If the cull time is high, the runtime system has less time to draw because it is culling groups in the database hierarchy that are too small. Conversely, if the drawing time is high, the runtime system has less time to cull because it is drawing large groups.

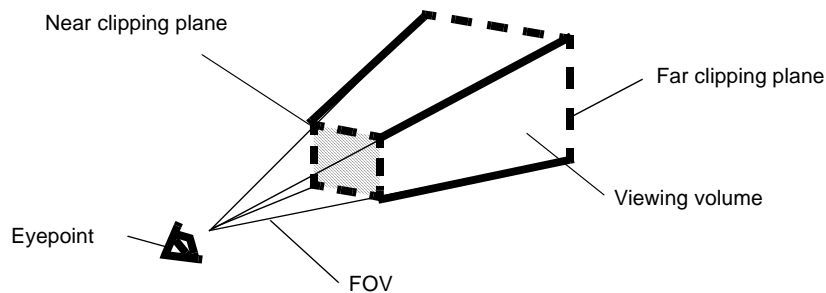
To balance culling and drawing, adjust the size of your groups. Increase group sizes while eliminating other groups in the database hierarchy if the runtime system spends too much time culling. Decrease group sizes by arranging more groups if the runtime system spends too much time drawing. An entire group must be redrawn if even a small area of the group is in the field of view, so it is a good idea to keep the size of groups as small as possible.

You can check the runtime system's statistics for culling, drawing, and application processing to compare to your simulation's requirements. See "Structuring the Hierarchy for Efficiency" on page 6-2 for more information about properly organizing elements in your database to optimize the cull and draw stages.

Defining a Viewing Volume

In most runtime systems, you must define a *viewing volume*. The viewing volume determines how an object is projected onto the screen, such as a *perspective* or *orthographic* projection. With a perspective projection, the farther an object is away from the eyepoint, the smaller it appears in the final image. Objects in a database scene have depth. With an orthographic projection, the sizes of objects do not change when the distances from the eyepoint change. Perspective is not represented with an orthographic projection, and objects in the database scene appear flat. The viewing volume also defines the objects or parts of objects that are clipped from the final image.

The viewing volume resembles a *frustum* of a pyramid (a truncated pyramid with its apex removed) formed by the horizontal and vertical field of view (FOV) and two parallel clipping planes in the front and back, as shown in the following illustration. The eyepoint lies at the apex's original position.



For each rendered frame, objects that fall within the viewing volume are projected toward the eyepoint. Objects that are closer to the eyepoint appear larger than objects that are farther from the eyepoint.

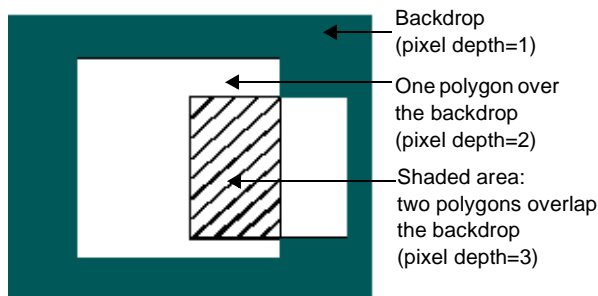
The viewing volume clips objects that lie outside of it. Clipping planes are on the frustum's four sides, top, and bottom. The objects that are displayed within the confines of the clipping planes constitute the objects that are rendered for each frame.

The viewing volume that you define in Creator is not associated with the runtime system's viewing volume. The Creator viewing volume is meant to optimize viewing performance and emulate the runtime version of the Creator database. See "Moving Clipping Planes" on page 6-23 for information about adjusting clipping planes within the Creator viewing volume to increase viewing performance.

Depth Complexity

Polygons are converted into pixels so that the runtime system can draw and display them on the screen. *Depth complexity* refers to the number of times each pixel is drawn per frame. Since all images included in the viewing volume are drawn, including backdrops, pixels within images that overlap are drawn more than once.

Areas of the scene with no overlapping images have a depth complexity of 1, and areas of the scene with overlapping images have a depth complexity equivalent to the number of polygons in the same screen space. The following figure illustrates this concept.



As depth complexity increases, the time that the runtime system needs to draw pixels and meet target frame rates increases as well. Check your platform's technical specifications for the *pixel fill rate* to find the number of pixels that your runtime system can draw per second.

To reduce depth complexity, you can:

- Remove faces that are hidden by other faces and are never visible.
- Remove texture underneath buildings or areal features.
- Remove extra *subfaces* using the **Cut Subfaces** tool. Subfaces lie on top of other faces and have nodes that are placed underneath their parent nodes in the database hierarchy. Subfaces are used on systems that cannot resolve coplanar faces, such as z-buffer systems.

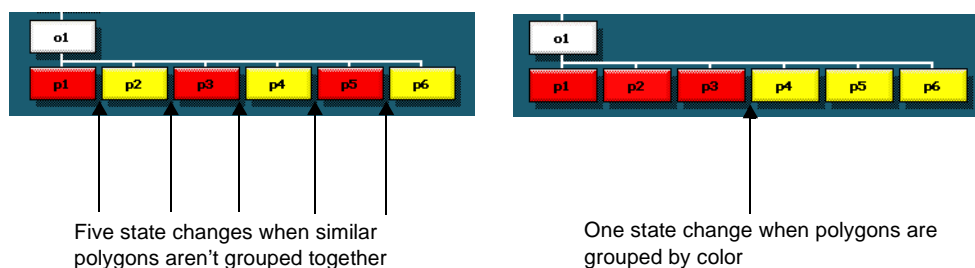
The **Cut Subfaces** tool merges subfaces with their parent faces and retriangulates the results. The number of polygons increases, but the pixel fill rate decreases.

For more information about subfaces and z-buffer systems, see “Z-Buffer” on page 6-14.

State Changes

State changes occur when a polygon with one set of attributes or *states*, such as color, material, texture, and lighting type, switches to another polygon with different states in the database hierarchy. This switch, or state change, causes the runtime system to pause while loading a new set of attributes before drawing the next polygon. You can reduce state changes and speed up the draw stage by grouping polygons with common attributes within an object.

Groups of nodes are useful for integral models with similar parts. For example, you can group all polygons with the same color or texture together.



Moving Nodes in the Hierarchy

By default, nodes are arranged as you create them and are drawn in a *fixed-list* drawing order. Nodes are drawn in front of their preceding nodes as the database hierarchy is traversed. If you manually move the nodes to new positions in the hierarchy to avoid state changes, the new order of nodes can affect how the geometry is drawn. For example, polygons could be hidden by other polygons that are drawn on top.

When you change the order of nodes in the hierarchy, you should also view them in the runtime version of your database. For more information about fixed-list and other drawing orders, see “Draw Order” on page 6-7.

4 *Exploring Modeling Techniques*

Now that you understand the fundamentals of how Creator works, you can explore some basic modeling techniques to help get you started. For realtime interactive applications, you want to create low-polygon models while still maintain a high degree of realism. The Creator tools help you achieve this goal, if you understand the techniques to use.

For basic modeling, you start with the tracking plane for plotting points and constructing polygons. To add realism, you can create effects with textures, lighting, shading, and sound. You can also animate models using Degree of Freedom (DOF) nodes, which add movement to the geometry below it in the database hierarchy.

Using the Tracking Plane

When you draw a figure in a 2D drawing program, it is assumed that you are drawing on the surface of a screen. When you draw a rectangle, for example, you might perceive that its vertical edges represent its height and its horizontal edges represent its width. In 2D, there is no dimension to represent depth. The depth is assumed to be at the surface, as though you are drawing on paper.

A 3D modeling environment has an extra dimension for depth. The rectangle might look exactly the same, but you would not be able to give its location in 3D space because the number of possible coordinates for depth is infinite. To constrain the third dimension to a single plane as you draw, Creator provides a *tracking plane*. The tracking plane is invisible, but a visible 2D grid is superimposed on it. The 2D grid not only lets you see the tracking plane, but it also lets you measure geometry as you draw it.

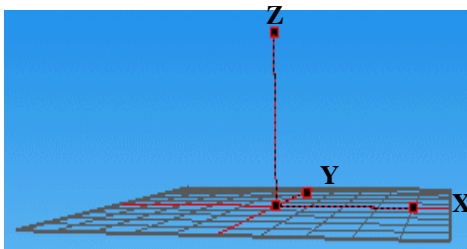
Creator projects points entered with the mouse into the 3D coordinate system on the tracking plane. The tracking plane ensures that you draw your geometry correctly in three dimensions. You can create simple shapes on the tracking plane, but to build off of those shapes, you must first reposition the tracking plane. By moving the tracking plane along different dimensions as you model, you can carefully analyze how to construct faces. For example, you can decide if you want to slice faces, combine faces, reshape faces, or simply draw new faces.

To ensure accurate modeling and editing, you can use the tracking plane for:

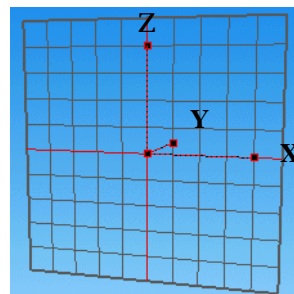
- Aligning the tracking plane along the x - y axis, y - z axis, or x - z axis
- Placing items onto other geometry
- Slicing geometry along a plane
- Modifying geometry
- Mapping textures onto geometry
- Creating planar and nonplanar faces

Positioning the Tracking Plane

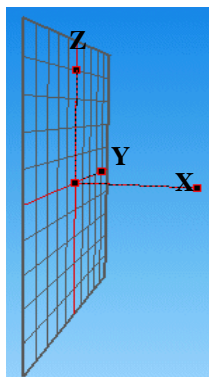
To begin modeling, you can position the tracking plane on the x - y coordinate axis to constrain the z dimension, on the y - z coordinate axis to constrain the x dimension, or on the x - z coordinate axis to constrain the y dimension using controls in the **View** panel. The following illustration shows the tracking plane aligned along these axes.



X-Y Coordinate Axis



X-Z Coordinate Axis



Y-Z Coordinate Axis

The Creator modeling environment is based on a global database system where the default center of the grid (0,0,0) or *absolute center*, is at the center of the database universe. When you use grid controls in the **View** panel to reorient the tracking plane to a face or other area in the database, the center of the tracking plane is placed at the new vertex that you select, which is called the *relative center*. If you press the coordinate plane buttons XY, XZ, YZ in the **View** panel, the grid's center defaults to the center of the database. You can rotate the grid and tracking plane along any of the axes using the Rotate Grid control to keep your tracking plane definition. *Roll* rotates the tracking plane around the *y*-axis, *pitch* rotates the tracking plane around the *x*-axis, and *yaw* rotates the tracking plane around the *z*-axis.

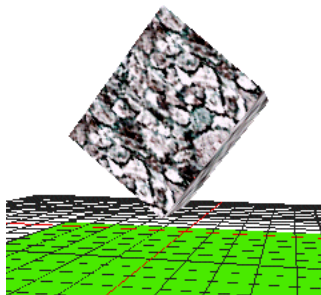
Placing Items Onto Other Geometry

You can use the tracking plane to place items onto other geometry with precision and control. In this way, you can position geometry accurately along curved or sloping surfaces, for example.

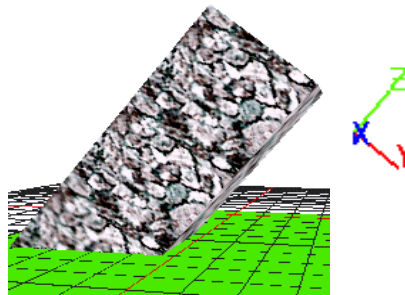
The tracking plane can be aligned along a face using the **Trackplane from Face** tool or up to three vertices using the **Trackplane from Vertex** tool in the **View** panel. The Creator online help lists the steps for using these tools.

After the tracking plane is aligned where you want to place your geometry, you can use different techniques and tools to move the item onto new geometry. For example, you can either create new geometry on the realigned tracking plane, use the **Translate** tool in the **Maneuver** toolbox to translate the item to a certain point on the tracking plane, or use the **Plant** tool in the **Modify Geometry** toolbox to plant the lowest vertices (bottom

of model, for example) to the tracking plane surface. The **Plant** tool plants in the direction of the z-axis, as shown in the following example.



You can use the **Plant** tool to plant a model to the tracking plane surface



After you click the **Plant** tool, the lowest vertices of the model plant to the tracking plane in the z-axis direction

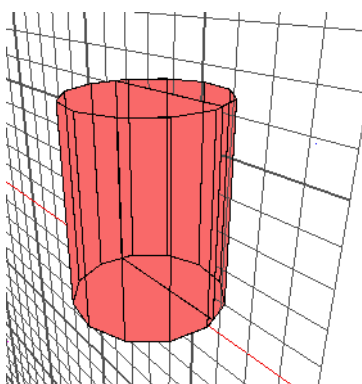
See Chapter 3, “Construct Your First Realtime Model,” in the *Desktop Tutor* for a lesson on making a chimney and planting it on a roof.

Slicing Geometry Along a Plane

You can slice geometry along the tracking plane. The **Slice** tool in the **Modify Geometry** toolbox uses the tracking plane as an even surface to split all selected faces that intersect the tracking plane. The tracking plane can easily be set at any location on a piece of geometry.

After you set the tracking plane to the location where you want to slice using the **Trackplane from Face** tool, for example, you click the **Slice** tool. The original geometry is automatically deleted, and the new geometry that is sliced is attached to the original parent in the database hierarchy. The sliced face nodes retain the original node names, but have an underscore and integer appended to the names, such as p1_1, p1_2, or Face_1, Face_2. This is done so that you can locate the sliced face nodes in the database

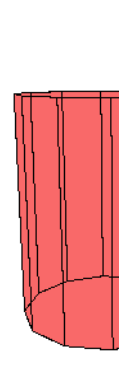
hierarchy. You can then delete the sliced nodes to see your geometry sliced in the Graphics view.



You set the tracking plane where you want to slice the geometry, and click the **Slice** tool



The sliced face nodes are renumbered and attached to the original parent



After you slice with the **Slice** tool, you can delete geometry that you do not want

See Chapter 3, “Construct Your First Realtime Model,” in the *Desktop Tutor* for a lesson on using the tracking plane to slice an extended roof.

Modifying Geometry

You can use the tracking plane to help adjust the height or size of geometry using the **Project** tool or **Scale** tool, for example. With the tracking plane, you can select a specific measurement instead of using an approximation.

To raise a face to a specific height, you first need to position the tracking plane to that height by setting the tracking plane Grid Offset in the **View** panel to a certain number of units. Then, you can select the vertices on the face in Vertex mode and click the **Project** tool in the **Modify Geometry** toolbox. This tool snaps the selected vertices that

are closest to the tracking plane to the tracking plane level. This method is useful when you want to raise a series of faces to the same level at the same time.



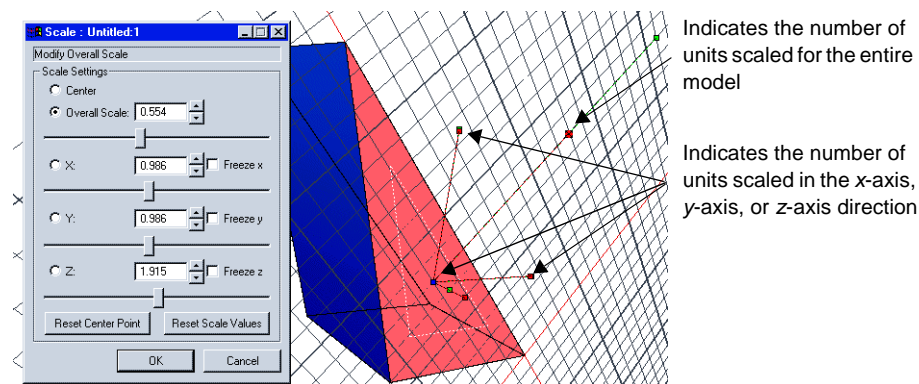
You position the tracking plane over the top of the model to raise its roof



After you click the Project tool, the roof is raised to the tracking plane level

See Chapter 3, “Construct Your First Realtime Model,” in the *Desktop Tutor* for a lesson on using the tracking plane with the **Project** tool to raise the peaks of roof gables to a certain height.

To scale geometry to a certain measurement, you can choose the **Trackplane from Face** tool in the **View** panel to first align the tracking plane to a face. Then, select the **Scale** tool in the **Maneuver** toolbox to increase or decrease the size of the model in either the x-axis, y-axis, or z-axis direction. You can also scale the model in all directions at once. The tracking plane can be used as a guide if you use the slider in the *Scale* dialog box to adjust the sizes of the faces.



Indicates the number of units scaled for the entire model

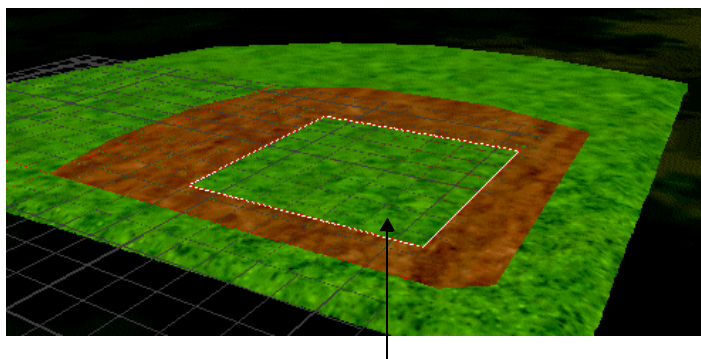
Indicates the number of units scaled in the x-axis, y-axis, or z-axis direction

See Chapter 3, “Construct Your First Realtime Model,” in the *Desktop Tutor* for a lesson on using the **Scale** tool and tracking plane to extend a roof.

Mapping Textures to Geometry

You can align the tracking plane to a face using the **Trackplane from Face** tool in the **View** panel and map a texture to it. The tracking plane is useful when you want to map a large scale terrain texture, for example. You can select vertices on the grid as the coordinates for mapping.

Using a **Put Texture** tool in the **Texture** toolbox, you choose vertices on the tracking plane as the **To** points (origin, alignment, and shear) in the *Put Texture* dialog box. Each **From** point on the texture pattern maps to the corresponding **To** point in the database. The online help has detailed steps for using the **Put Texture** tool.



You can align the tracking plane to a face to correctly map a terrain texture to the face

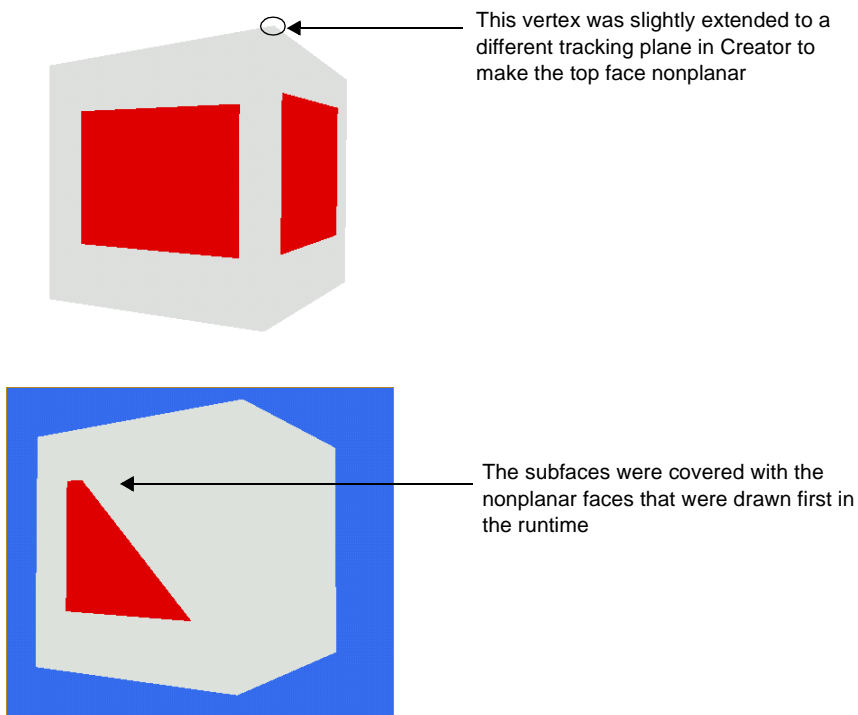
See Chapter 7, “Applying Textures to Your Farmhouse,” in the *Desktop Tutor* for a lesson on using the **Put Texture** tool and the tracking plane to map a texture onto a roof.

Creating Planar and Nonplanar Faces

When you create models for a realtime simulation, you want to be sure that your models will appear in the runtime as you intend. Faces should be drawn in the correct order and not hide other faces. A *planar face*, which has all vertices on the same tracking plane, will render correctly in the runtime. Even though the runtime system triangulates *nonplanar* faces, in which at least one vertex lies outside the tracking plane, it is best to use the tracking plane as you create your models to ensure that faces are planar.

The effects are most noticeable when you rotate a nonplanar face. The first illustration shows a grey cube with a red *subface* on each face. A subface is a nested face attached to another face in the hierarchy. A corner on the top face of the cube was extended to a different tracking plane to become nonplanar. The two side faces that join the corner also become nonplanar.

The second illustration shows the cube in Vega, which is MultiGen-Paradigm’s runtime software. As the cube was rotated in Vega, the *z-buffer* processed the extended portion of the cube first that covered parts of the red subfaces. The z-buffer algorithm computes the distance (z value) from the eyepoint to each pixel, stores this distance in the z-buffer, and draws the closest pixels on top of the others. The runtime version correctly rendered the planar faces and nonplanar faces, but did not produce the same cube that was modeled in Creator. The cube was incorrectly modeled with nonplanar faces in Creator for the desired effect in the runtime.



Faces in your database can become nonplanar in a number of ways. Imported models from other modeling software can have nonplanar faces. Also, if you translate a vertex

from a face to a different area in the database to change its shape, for example, and you do not first align the tracking plane to the face, the face can become nonplanar.

To correct nonplanar faces, you can quickly find and select them in your database by choosing **Select/Select Nonplanar Faces** and flatten them to the tracking plane with the **Project** tool in the **Modify Geometry** toolbox. See the **Project** tool description in the online help for detailed steps.

You can also use the **Triangulate** tool in the **Modify Face** toolbox to triangulate nonplanar faces. The **Triangulate** tool divides nonplanar faces into triangles so that they are planar. See the **Triangulate** tool description in the online help for detailed steps.

Modeling with Polygons

You build your models as a collection of shapes on the tracking plane. Creator provides **Face** tools so that you can create polygons that resemble rectangles, circles, or your own design. You can also use a variety of tools and techniques, such as slicing faces or rounding edges, to modify and enhance the shapes. As you create your polygons, keep these tips in mind so that they render correctly and efficiently:

- *Create convex polygons.* As defined in “What are 3D Graphics?” on page 1-3, *convex* polygons have no indentations, and lines joining any two points in the interior of the polygon also lie in the interior. OpenGL tessellates convex polygons much easier than concave polygons. You can use the **Split Face** tool in the **Modify Face** toolbox to split a concave polygon into convex polygons, as shown in the following illustration. Refer to the **Split** tool procedure in the online help for instructions on using this tool.



Concave polygon

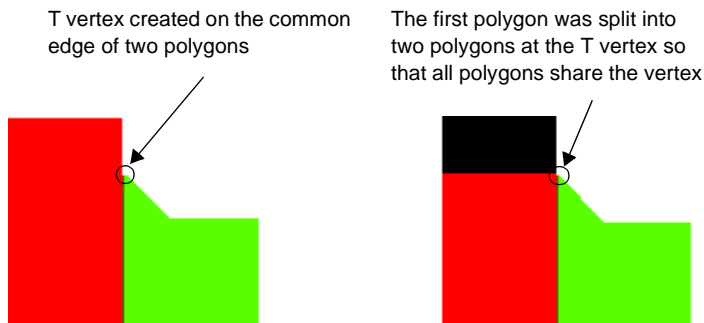


Same shape split into two convex polygons

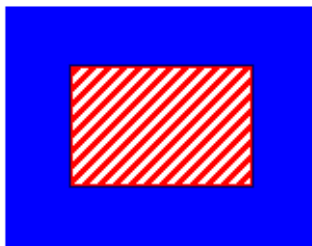
- *Create coplanar vertices.* As discussed in “Creating Planar and Nonplanar Faces” on page 4-7, all vertices on a face should lie on the same plane (coplanar vertices) to render correctly. Otherwise, nonplanar faces can hide other faces or holes be-

tween nonplanar faces can appear in your model as it rotates in the runtime. The **Face** tools create planar faces for you, but be sure to align the tracking plane along a face if you want to move one of its vertices to a different location.

- *Avoid T vertices.* T vertices are areas where two or more adjacent polygons share an edge, and the polygons do not share a common vertex on that edge, as shown in the following illustration. Cracks can appear along the common edge. To correct T vertices, split the face that does not share the vertex at the T vertex location using the **Split** tool.



- *Avoid coplanar faces.* A coplanar face is a polygon that lies directly on top of another polygon, as shown in the following example. Z-buffer fighting can occur when a z-buffer system cannot resolve which polygon to display on top. You can change one of the coplanar faces to be a subface of the other in the hierarchy so that the runtime can draw the faces in the correct order. See “Z-Buffer” on page 6-14 for more information about changing coplanar faces on a z-buffer system.



Coplanar faces

Applying Textures

Texture patterns are bitmapped images that are applied to polygons to give your database a photo-realistic appearance without increasing the polygon count. Creator supports these types of texture patterns:

- Intensity (one component: 8-bit greyscale)
- Intensity-alpha (two components: 8-bit grayscale, plus 8 bits of transparency)
- RGB (three components: 8 bits each of red, green, and blue)
- RGB-alpha (four components: 8 bits each of RGB plus 8 bits of transparency)

Creator provides many tools for applying and modifying textures. Using the **Insert Texture** tool in the **Properties** toolbox, you can apply the texture that is in the **Texture** palette with its current mapping properties. You can map a texture as well as modify it with the tools in the **Map Texture** toolbox. Textures can be scaled, rotated, or blended with other textures for different effects. You are only limited by the amount of texture memory in your runtime system.

The tools in the **Map Texture** toolbox apply texture patterns to faces and modify the mapping of applied texture patterns. For example, you can choose how a texture is mapped on different faces using the **Put Texture** tools. Before you apply textures, it is helpful to understand how texture mapping works.

Texture Mapping

Texture Mapping is the process of applying a texture pattern onto one or more polygons (faces). When OpenGL renders a texture pattern in Creator, each *texel* is mapped to a *pixel* on the computer screen. A texel, which is an abbreviated word for texture element, is a dot that makes up a texture pattern. A pixel, which is an abbreviated word for picture element, is the smallest display unit of a video image.

More precisely, OpenGL first converts the x,y coordinate of the texel to an address, called a u,v coordinate. The texture is applied to a face using the u,v coordinates, which are stored in the face's vertices. In Creator, you can see a texture's u,v coordinates that are mapped to a face's vertices in the *Vertex Attributes* window.

The size of the texel is determined by the resolution of the image and its scaled size. The number of colors that a texel represents is determined by the number of the texel's color components (1 byte each) and the texture pattern.

Depending on the scale of the image, it could be said that one texel is equivalent to one pixel. If you need to either increase or decrease the size of your texture before mapping it, you can either magnify or minimize your texture in Creator using the **Minification** and **Magnification** tools in the *Texture Attributes* window. When you minimize the texture, several texels map to one screen pixel; when you magnify the texture, each texel maps to several screen pixels. The **Minification/Magnification** texture filters soften, harden, or blend texels together for a more realistic effect on an object as the eyepoint changes.

The Put Texture Tools

Creator provides two **Put Texture** tools (**3-Point Put** and **4-Point Put**) that let you control how a texture is mapped on a face. The **3-Point Put** and **4-Point Put Texture** tools let you choose points on the image that map directly to points on the face.

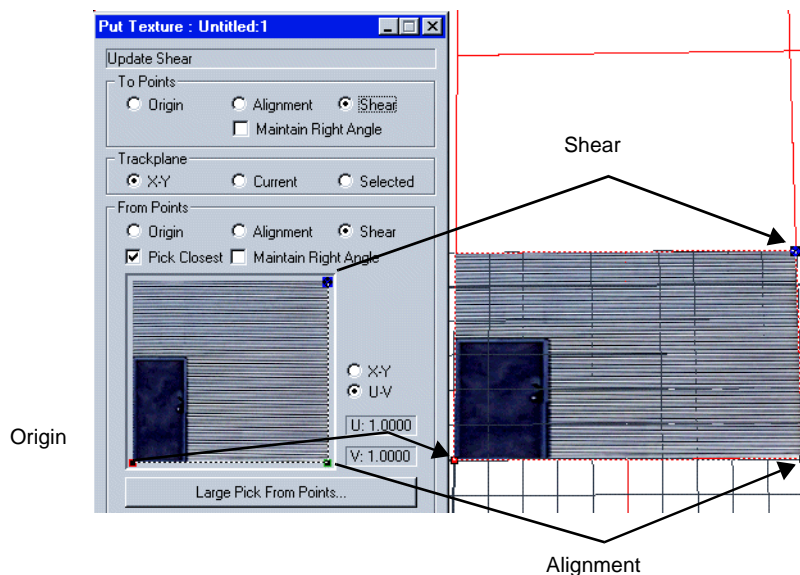
The standard **Put Texture** tool is also referred to as the **3-Point Put** tool because you map three points from the texture's image in the *Put Texture* dialog box (origin, alignment, and shear points) to a face's origin, alignment, and shear points. The **From** points can either be x,y coordinates (in texels measured from the lower-left corner of the texture pattern) or u,v coordinates anywhere on the texture pattern. A grid space in the **Map Texture** tool is equal to one u,v unit.

To choose the **To** points on the face, you choose the face's **Origin**, which corresponds to the texture pattern's lower left corner by default, **Alignment point**, which defines the alignment and scale of the texture pattern along the u -axis, as well as the **Shear point**, which defines the alignment and scale of the texture pattern along the v -axis. These points lie on a texture pattern grid that you can rotate around the face for placement.

Note: You must use the middle-mouse button to map the texture to a face. If you use the left-mouse button, you map the texture to the tracking plane instead.

If you hold down the Alt key while entering the Alignment and Shear points, the texture pattern is not scaled. Instead, the length of the u -axis for the Alignment point

and the length of the *v*-axis for the Shear point are defined using the texture's **Real World Size** attribute, which is the width and height in meters.

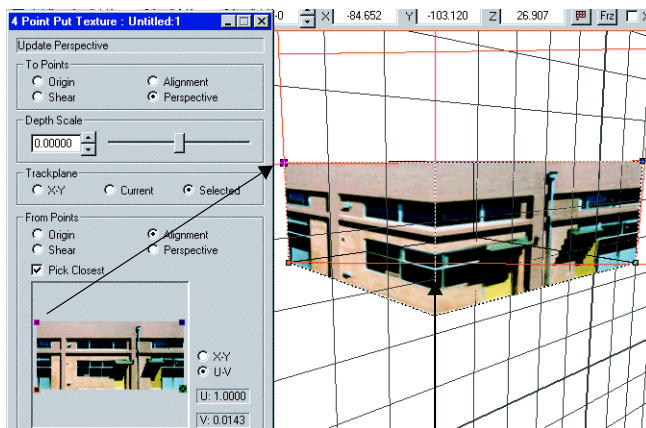


With the **3-Point Put** tool, you can map three **From** points on the texture pattern (origin, alignment, shear) to three **To** points on the face (origin, alignment, shear). You can position, orient, scale, and shear the texture pattern on the face while you apply it using the **From/To** points.

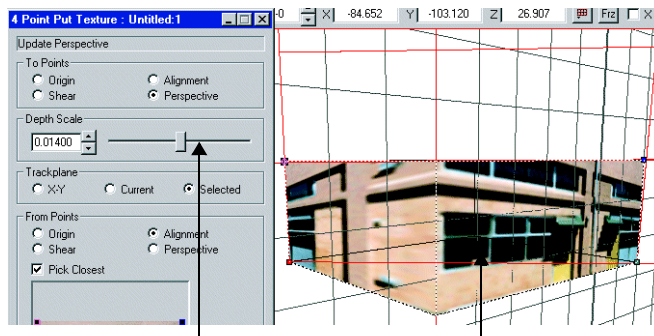
You use the **4-Point Put Texture** tool in the same way except you also define a *Perspective control point*. The Perspective point changes the perspective of the texture pattern to accurately align with geometry that does not face the front, such as the side of a building. Instead of rotating the faces to match with the texture pattern, you can adjust the perspective of the texture pattern to match the face's direction instead. With the **4-Point Texture** tool, you can map a texture with precision.

The **4-Point Put Texture** tool is also useful for aligning a texture with a corner. After you have applied a texture to cornered faces, you can use the **Depth Scale** tool to pull the middle of the texture away from its grid (defined by the origin, alignment, and shear points) to match the geometry of a corner. For best results, the dimensions of the

building should be scaled proportionally to the dimensions of the building in the image. The following example illustrates this method.



Using the **4-Point Put** tool, the Perspective point was used to correctly map the texture to the side of the building. The window, however, falls on a corner.



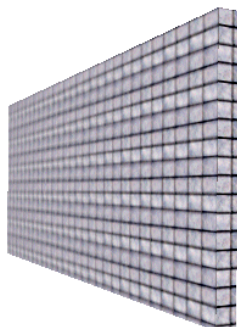
The **Depth Scale** tool was used to align the texture to the corner of the building so that the window falls on the side instead of on a corner. The building dimensions can be later scaled to match the dimensions of the image.

Note: See the online help for detailed instructions on using any of the **Put Texture** tools.

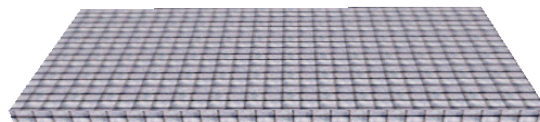
The Surface Project Texture Tool

The **Surface Project Texture** tool wraps a texture around selected faces or all sides of a volume without shearing the texture. The texture is applied to the first face in the selected object and then wrapped across contiguous linear polygons that share common edges. Non-contiguous polygons are arbitrarily mapped. The **Surface Project Texture** tool is especially useful for applying texture on walls or fences.

When you click the **Surface Project Texture** tool, you must specify a *Repetition Factor*. A Repetition Factor determines the number of times a pattern repeats across the surface in both the u and v directions. Valid values for the Repetition Factor are from .01 to 10,000; the larger the number, the smaller the pattern.



The **Surface Project Texture** tool wraps a texture around the main face and contiguous linear polygons without shearing



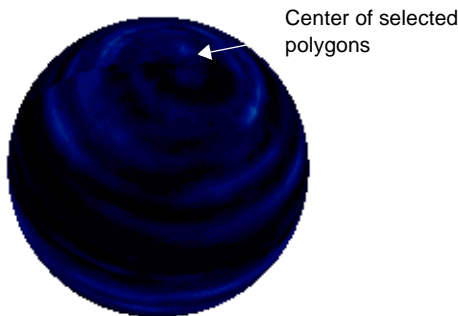
The bottom face of the wall, which is a non-contiguous polygon, is arbitrarily mapped

The Spherical Project Texture Tool

To wrap a texture around a sphere, use the **Spherical Project Texture** tool. The **Spherical Project Texture** tool covers a continuous surface with a smooth flow of texture by projecting the pattern spherically out from the center of all selected polygons.

When you click the **Surface Project Texture** tool, you must specify a *Repetition Factor*. A Repetition Factor determines the number of times a pattern repeats across the surface

in both the u and v directions. You can also add the texture mapping you define to the **Texture Mapping** palette, which is useful if you want to reuse the texture mapping.



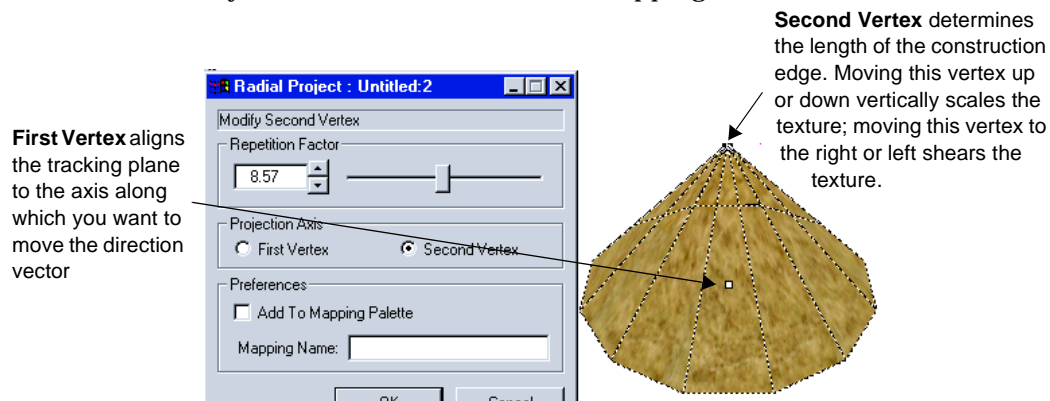
The **Spherical Project Texture** tool is used to map a texture to a sphere. It projects a pattern spherically out from the center of the selected polygons.

The Radial Project Texture Tool

The **Radial Project Texture** tool (cylindrical project) maps texture in a continuous flow around a surface by projecting a texture pattern outward from a *construction edge*. A construction edge is a temporary edge used for modeling purposes that is not saved with the database. You define a direction vector for the construction edge (with first and second vertices), which also defines the direction in which the texture is projected. The length of the construction edge scales the texture pattern along the v -axis. The **Radial Project Texture** tool is typically used to wrap texture around a cone, cylinder, or cube.

When you click the **Radial Project Texture** tool, you specify a *Repetition Factor*. A Repetition Factor determines the number of times a pattern repeats across the surface in both the u and v directions. You also select two vertices that define the direction

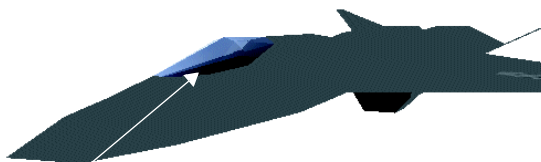
vector. You can add the texture mapping you define to the **Texture Mapping** palette, which is useful if you want to reuse the texture mapping.



The **Radial Project Texture** tool maps texture in a continuous flow around a surface by projecting a texture pattern outward from a construction edge that you define

The Environment Map Texture Tool

Environment mapping in Creator is a type of “reflection mapping” that simulates a reflective surface like glass or water. The **Environment Map Texture** tool maps the current texture to a face and adds a mirror-like effect to the texture. As you rotate the object, the mirror-like effect is enhanced. The texture’s u, v coordinates continually remap to the object’s pixels as the object and eyepoint move, creating an illusion of a change of reflection. OpenGL uses the u, v coordinates as an address to map a texture’s texels to pixels on the object to render the drawing.



The **Environment Map Texture** tool was used to map a cloud texture to an airplane’s canopy. The texture was blended with a transparent material so that the airplane cockpit is visible through the canopy.

You can apply shading, such as Gouraud, to an object to blend with the mapped texture if the blending attribute of the texture is on. Shading is useful for applying tints to windows, for example. As the object moves, the shading will change since shading is calculated according to the intensity of the light at a certain position.

To create the illusion of a reflected object in Creator, you can take a picture of the object using a graphic editing tool, such as Adobe Photoshop, and import it into the **Texture** palette as a texture. You can then apply it to a surface as any other texture using the **Environmental Map Texture** tool. The surface will appear to be reflecting the objects in front of it, such as a mirror or a lake.

Truly accurate reflective mapping involves a ray-tracing program, commonly used in other modeling tools, that can create a much more realistic reflection. Pixels on a mirrored surface are assigned colors based on objects that rays hit, and as a result, objects that move around are accurately mirrored. Creator does not use ray tracing because it requires a lot of mathematical computation and slows down performance. A ray-tracing program is effective for animation, but inefficient for realtime applications.

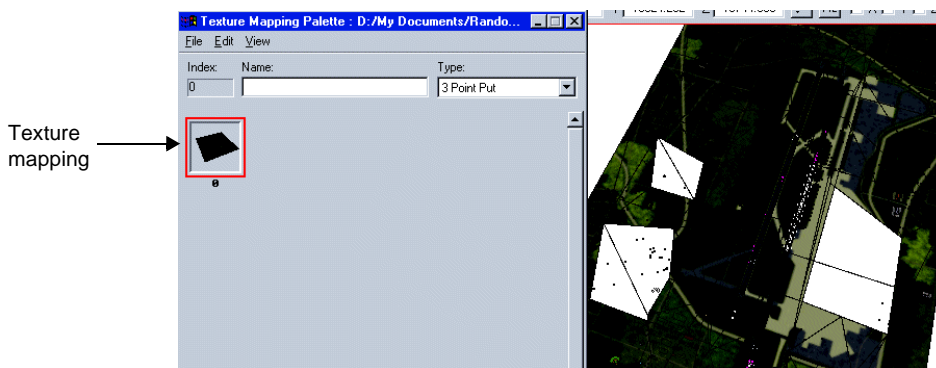
Other Texture Techniques

If you have a large database with a lot of polygons to texture, you want techniques that help you select polygons and save time in the texturing process. You also want to add detail and other effects without adding to your polygon count. You can save time by using the **Texture Mapping** palette and selecting multiple faces to texture. If you are using levels of detail (LOD) in your database, you can apply textures to more than one LOD at once. The MultiTexture feature can help to add detail without adding polygons.

Using the Texture Mapping Palette

The **Texture Mapping** palette is useful for storing a previously mapped texture and its coordinates. You can quickly apply this texture mapping to many faces. This method is useful when you have a large area to cover with a specific texture. Instead of selecting each face and using a **Put Texture** tool to choose the coordinates for mapping the texture, which can be time-consuming for large databases, you can choose a texture in the **Texture Mapping** palette with its predefined alignment and select the faces on which

to apply it. The current texture is applied to the faces when you press Ctrl and click the **Insert Texture** tool.



The **Texture Mapping** palette is useful for covering many specific areas of a large-scale terrain. Instead of using a **Put Texture** tool to apply texture to each face, you can apply the texture once, save the mapping, and quickly reapply it to different faces.

A texture mapping is useful when you want to modify a texture after you apply it. You can apply a texture once, save the mapping in the **Texture Mapping** palette, and continue to reapply the texture mapping from the **Texture Mapping** palette. If you adjust the texture mapping, all of the textured faces with the mapping update with the change.

The texture is aligned with the same u, v coordinates on all of the geometry, however, and can appear distorted if all of the geometry is not the same size and shape. The

following texture was applied to the first face but misaligned on the smaller face next to it with the same mapping coordinates.



The texture was aligned on this face and saved in the **Texture Mapping** palette



The same texture was mapped onto a smaller face with the same mapping attributes and was misaligned

If you adjust the texture map in the **Texture Mapping** palette, all of the textured faces are adjusted in the same way. After you adjust the texture map, you should check all affected faces to see if the texture is correctly aligned.

Applying Textures to Multiple LODs

You can apply a texture to multiple LODs at once. LODs, as described in “Using Levels of Detail” on page 6-17, are sets of models that represent the same object with varying amounts of complexity. LODs are used for viewing objects at different distances. It is convenient to apply a texture to multiple LODs when you have a lot of LODs that you want to add a texture to.

In the database hierarchy, you can select all LODs by selecting the parent node (usually g1) and selecting the **Toggle Display** tool in the **Hierarchy** toolbox. All LOD nodes are selected in the Hierarchy view, but the model is not visible in the Graphics view. To see the model and keep all LODs selected, you can either press **H** or select the **Toggle Display** tool again. You can use a **Map Texture** tool to apply the texture.

If you want to apply a texture to specific faces in several LODs, you can select the nodes in the database hierarchy while pressing the Shift key. The faces remain selected in the different LODs so that you can apply texture to them at one time.

You can also search for specific geometry by its node attributes. You can choose **Attributes/Attribute Search** to search your database and select only the faces with attributes that you specify in the *Attribute Search* dialog box. For example, if you want to apply texture to only terrain in a database populated with terrain and culture, you can set the Terrain flag to *True* in the *Attribute Search* dialog box.

Applying MultiTextures

The MultiTexture feature lets you map more than one texture to a face. With multitextures, you can blend multiple textures together for interesting effects without adding additional polygons for detail. You can also blend a high-resolution image, such as an airport, with a low-resolution image, such as surrounding terrain that will be viewed in the distance.

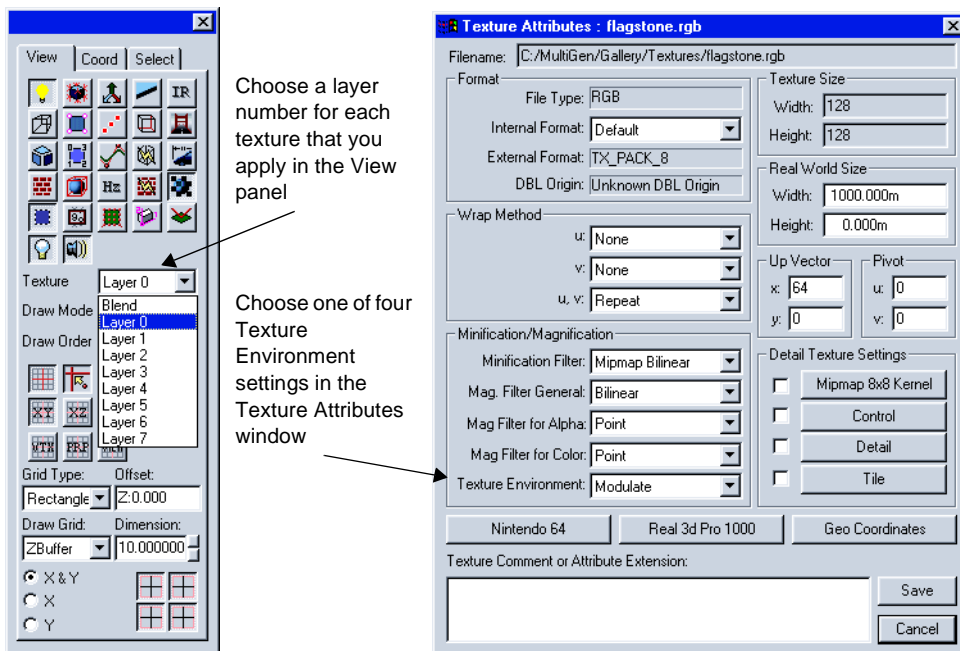
You can map up to eight textures: one base and seven layers, depending on the number of layers that your graphics card can support. Check your video card specifications before you apply multitextures.

This is the basic process for applying multitextures:

- Select the face to which the layer will be applied in your database.
- Choose a layer number in the **View** panel. Layer 0 is usually for the base texture and Layers 1-7 are usually for additional textures. You can apply the textures in any order, however.
- Apply the texture to the face with any **Map Texture** tool.
- Choose another layer number in the **View** panel, and apply a different texture to the same face. You can add up to seven additional textures as layers on the face.

Select the Blend function in the **View** panel to see the blended textures. You can also blend textures with other textures or polygon colors using the Texture Environment

algorithms (**Modulate, Blend, Decal, Replace**) that you select in the *Texture Attributes* window.



Texture Environment Settings

The Texture Environment settings define how a texture interacts with the color of the face or object to which it is applied and can be any one of the following:

- **Modulate** integrates texture values with polygon color. In this mode, face color affects the colors of certain kinds of texture patterns applied in the database. This is the default setting.
- **Blend** modulates the primary and alternate color of the polygon based on the given texture element intensity value. For example, if a texel has a greyscale value of 26 (10% of the range 0 to 255), 10% of the polygon's alternate color blends with 90% of the primary color.
- **Decal** replaces the polygon color with the texture color. An RGBA texture blends with polygon colors in a ratio determined by the texture's alpha component. The

polygon's alpha component is unchanged. Decal is only applicable to RGB and RGBA texture patterns.

- **Replace** replaces the polygon color with the texture color. An RGBA texture replaces polygon colors.

See Chapter 7, “Applying Textures to Your Farmhouse,” in the *Desktop Tutor* for a lesson on blending a texture with the outside wall of a house.



In these examples, the MultiTexture feature was used to blend two different textures together on one face

Note: The u, v values for multiple textures will be the same if you map each texture to the same vertices. The texture's u, v values that are stored in a face's vertices appear in the *Vertex Attributes* window.

Reducing Texture Memory

Creator supports textures of any size, depending on what your graphics card can handle. The size of the texture that you can use also depends on your runtime system's memory and performance limits. The mapping of texels to pixels requires *texture memory* and a *pixel fill rate*. Texture memory is the amount of memory available for loading and storing textures. The pixel fill rate is the number of textured, anti-aliased pixels-per-second that your hardware can support. More memory is required for a texture with a lot of detail or for texturing a lot of polygons. Before you add texture, it is a good idea to check your runtime system's texture memory and pixel fill rate.

Creator can support textures that are larger than 4096 texels x 4096 texels, but most textures are 512 texels x 512 texels or less. To calculate the amount of texture memory that a texture would require, you can use this formula:

File size = X texels * Y texels * # color components

For example, if you have a 4096 x 4096 RGB texture,

File size = 4096 x 4096 x 3 bytes (RGB's three color components) = 50 MB

To reduce the texture memory, you can:

- Reduce internal data formats
- Apply subtextures

These techniques are discussed in the following sections.

Reducing Internal Data Formats

To fit textures into texture memory, you can reduce the texture's size or reduce the texture's *internal format* resolution. Internal data formats specify how the color components are packed or changed into different byte configurations for performance, efficiency, or image quality based on the texture's format (intensity, intensity-alpha, RGB, and RGB-alpha). You can change the internal data format of a texture in the *Texture Attributes* window.

For example, a texture file that has an RGB file format is considered a three-component color texture format with eight bits each of red, green, and blue. If this texture file is assigned an internal format of TX_RGB_5, the 24-bit texture is packed into 16 bits, with five bits of red, five bits of blue, and six bits of green. If the internal data format of a 4096 x 4096 RGB texture is changed to TX_RGB_5, its original file size of 50 MB is reduced to the following:

File size = 4096 x 4096 x 2 bytes (5/8 bytes + 5/8 bytes + 6/8 bytes= 2 bytes) = 33 MB

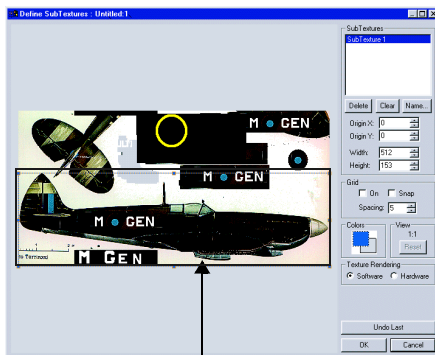
If the resolution is reduced, the drawing performance and paging capacity are improved while the image quality is slightly decreased. For flight simulations at high altitudes, a low-resolution texture may be fine. All of the internal formats that you can choose are described in the Creator online help.

Applying Subtextures

The subtexturing technique lets you select an area of a texture and apply it to a face. The selected area is the subtexture. You can save the subtexture in the **Texture Mapping** palette, which is used for storing a previously mapped texture and its coordinates. Subtextures save system memory by reducing the number of textures that are stored in the **Texture** palette.

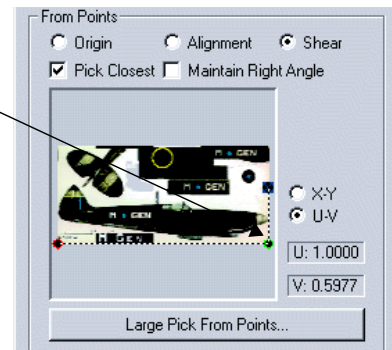
Subtextures are useful for applying sections of a texture to specific areas of a model. For example, you can load one texture into the **Texture** palette and apply sections of the texture to different parts of a house.

To define a subtexture, you choose **Palettes/Define Subtextures**, and fence-select an area on the main texture in the *Define Subtextures* window. Vertices on the subtexture outline are saved as **From** points, which the **Put Texture** tools use to map to faces. When the subtexture is the current texture and you open the *3-Point Put* window, for example, the **From** points that correspond to the subtexture are already selected. This is an easier process than manually selecting **From** points on a main texture each time you open it.



You fence-select an area on the main texture to define a subtexture

From points on the area you define as the subtexture are saved for the **Put Texture** tools



In the *Put Texture* windows, you can assign a name to the subtexture and add it to the **Texture Mapping** palette. You use **Texture Mapping** palettes, as described in “Using the Texture Mapping Palette” on page 4-18, to quickly texture faces with a saved texture map without starting one of the **Map Texture** tools.

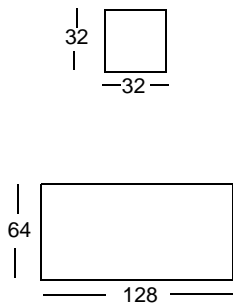
The information for each subtexture, such as the name, u, v coordinates, width, and height, is saved in the main texture’s attribute (.attr) file. A subtexture’s u, v coordinates

are set with respect to the main texture, so you cannot repeat a texture across a face. See the Creator online help for detailed instructions on applying subtextures.

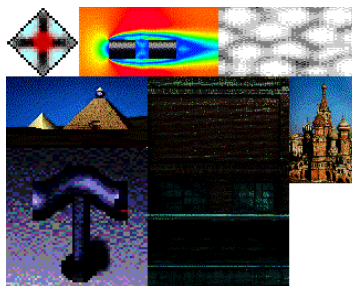
Creating a Subtexture Collage

You can create a single texture of individual texture patterns, or a texture collage, that you can load into the **Texture** palette. Each texture pattern can be fence-selected and used as a subtexture. This is useful if you need to apply several different texture patterns, but you want to conserve texture memory and reduce the number of textures that you load into the **Texture** palette.

The texture collage is created outside of Creator using Adobe Photoshop, the GNU Image Processor (Gimp), or another graphic editing tool. The subtextures must all be of the same type, such as RGB, and they must all contain the same texture attributes. The collage size can range from 8 to 256 texels, but the length of each side must be a factor of 2 (8, 16, 32, 64, etc.). To be most efficient, the subtextures should completely fill the collage without any unfilled spaces or borders. A collage of textures can become larger than what your video card supports, however, so check your video card specifications before you create a collage of textures.



Collage size must be a factor of 2



Avoid unfilled space in the collage

A Note on Graphics Card Support

Check your graphics card specifications for the number of textures that your graphics card can support. The number of textures that your graphics card can support often depends on the size of the textures. A card might be able to hardware-accelerate thirty-two 64 x 64 textures or two 256 x 256 textures, but can have trouble rendering four 256

x 256 textures. You can still render more textures than the hardware-accelerated number, but the drawing performance might be adversely affected.

With some PC graphics cards, it is better to have a lot of small textures because they can be quickly brought into card memory. With other graphics cards, too many textures can cause a performance penalty because a state change occurs with every new texture. Also, some graphics cards will only accept textures of certain sizes, for example, 256 x 256 or 128 x 128. Larger textures are automatically scaled down, which results in lower resolution textures.

Lighting and Shading Your Database

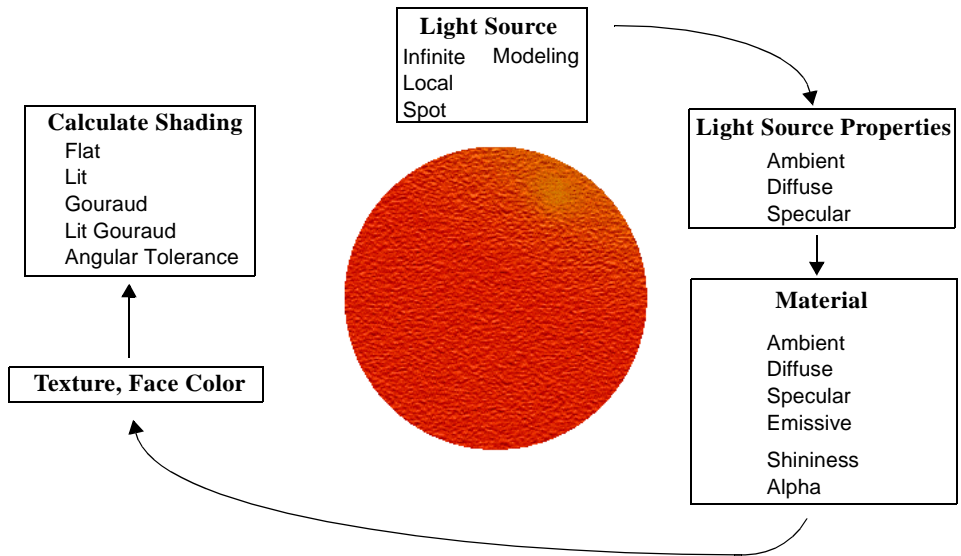
Lighting and shading help to add the 3D quality to your models. Without lighting and shading, a 3D model can look flat and its features can be hidden. They can also set a mood or create drama in your database.

To create lighting effects, you position light sources in the database, choose colors and other properties for the light, and shade faces. The Calculate Shading process uses vertex colors, as well as directions of vertex and light vectors to shade faces in a specific way. You can choose a shading model, such as Flat or Gouraud, with specific shading properties. After a face has been shaded, you can create more dramatic shading effects by modifying the shade attributes.

After you add lighting and shading to the database, you might want to add more character to your models to look realistic. Materials add luster, such as shininess or a glowing effect, to your models. You can also add materials to textured objects to enhance the face colors. Depending on the shading model that you choose, materials affect shading when Calculate Shading is performed.

You can add light points to simulate small points of light viewed in a distance, such as stars or city lights. Light points are colored vertices that do not illuminate a scene or

affect shading. You can choose to have a front color, back color, or both for each light point.



Creator includes these light sources, shading models, materials, and light points with properties to give your scene realism:

- **Infinite light source** - Creates general lighting, such as the sun, over the entire database. Infinite light sources have direction that you can change but no position that you set in the database.
- **Local light source** - Radiates light in all directions, such as a lamp, from a specific position that you set.
- **Spot light source** - Radiates light in a specific direction, such as a flashlight, from a specific position that you set.
- **Modeling light source** - Illuminates the entire database. Modeling light sources are attached to and move with the eyepoint. They are used for previewing models and are not exported to the runtime system. Modeling lights do not have a position that you set in the database.
- **Shading models** - Include specific properties that the Calculate Shading process uses to shade the database. After you set and position your light sources, Calculate Shading produces the lighting effects according to the shading model (Flat, Lit, Gouraud, Lit Gouraud) that you choose.

- **Materials** - Provide effects when combined with light and textures. Materials simulate light reflecting characteristics of substances like wood, plastic, and metal.
- **Light points** - Simulate points of light that are viewed in the distance, such as stars, runway lights, or city lights. Light points are vertices that can be adjusted and moved in the database. They do not radiate light or affect shading.

Light sources and materials use color properties (ambience, diffuseness, specularity, emissiveness) to simulate lighting effects. You define these properties for each light source and material that you add to your database.

The placements of your light sources and most of their effects are replicated in most runtime systems, but you should find out how your runtime handles these effects before you spend a lot of time applying lighting and shading in Creator. You might see different results in your runtime application than what you modeled. Modelers often approximate lighting and shading effects in Creator that realtime programmers refine in the realtime simulation.

Note: Sky colors (Day, Dawn, Dusk, Night), which determine the time-of-day in the Graphics view, are a modeling environment feature that are not considered lighting. You set sky colors in the **View/Sky Color** menu. Sky colors are not saved with the database.

Adding Light Sources

Before you add light sources to your database, it is probably easiest to first think about all of the lighting types that you want in your scene, such as infinite light from the sun and local light from lamps. Then, follow this basic procedure to light up your scene:

- Step 1** Define the types of light sources with their properties, such as color, in the **Light Source** palette (**Palettes/Light Source**). The types of lights are described in “Light Source Types” in this section.
- Step 2** Create and position light source nodes in the database hierarchy that reference one or more light sources in the **Light Source** palette. The light source nodes can either affect everything in the database or only the children of the light source nodes in the database hierarchy.
- Step 3** Move light sources to different areas in your scene using the *Place Light Source* window (**Attributes/Place Light Source**). You can place a light source on a model, such as a local light on a lamp. The default position for a local

or spot light source, which you can change, is at the database origin in the Graphics view.

Step 4 Set the direction of the light in the *Place Light Source* window. You can set the direction for infinite and spot light sources.

Note: Detailed procedures for adding lighting effects are in the Creator online help.

Light Source Types

Creator provides four types of light sources (infinite, local, spot, modeling) to simulate lighting that is in the real world. Creator renders up to eight light sources at one time in the database, but you can define more than eight light sources using light source nodes in your database hierarchy if your runtime system allows it. Each additional light source, however, increases the processing time.

To achieve the desired lighting effects, you define parameters for each light source type in the *Modify Light Source* window. In the **Light Source** palette, choose **Edit/Edit** or double-click the light source thumbnail to display the *Modify Light Source* window.

Infinite Light Source

An *infinite* light source acts as the sun: light is distributed onto all objects. It has direction but does not have a set position in the database. The default direction is at (0, -1, 0), which you can change.

You choose **Attributes/Place Light Source** to change the direction for an infinite light in the *Place Light Source* window. When the window opens, a graphical rotation guide appears in the Graphics view that you can rotate. As an alternative to rotating the rotation guide, you can enter values for the **Elevation**, which is rotation around the vertical axis, and the **Azimuth**, which is rotation around the horizontal axis.

Local Light Source

A *local* light source is located at a position in the database that you can change and shines in all directions, such as a lamp or candlelight. The default position is at (0, 0, 0), which is the center of the database. Like an infinite light source, an equal amount of light is distributed everywhere. Local light sources produce a more natural lighting effect than infinite light sources when lighting objects with sharp angles between faces, such as the corner of a room.

You choose **Attributes/Place Light Source** to change the position of a local light source in the *Place Light Source* window. When the window opens, a graphical rotation guide appears in the Graphics view that you can drag to different locations in the database. As an alternative to dragging the rotation guide, you can enter values in this window for the *x*, *y*, and *z* coordinates in the database.

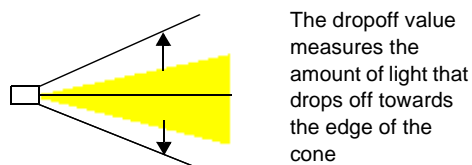
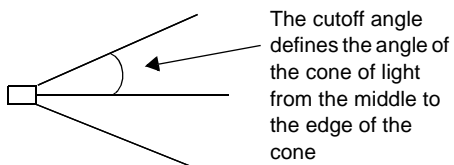
Spot Light Source

A *spot* light source shines in a specified direction from a specified position. It affects only the objects that fall within its directed cone of light. Spotlights are often used for lights such as streetlights or headlights on cars. The default position is at (0, 0, 0), which is the center of the database; the default direction is (0, 1, 0). You can change either the position, direction, or both in the *Place Light Source* window.

To open the *Place Light Source* window, choose **Attributes/Place Light Source**. When the window opens, a graphical rotation guide appears in the Graphics view that you can drag to different locations in the database. You can enter values for the *x*, *y*, and *z* coordinates in the database as an alternative to dragging the rotation guide. To change the direction, you can rotate the graphical rotation guide, or enter values for the **Elevation**, which is rotation around the vertical axis, and the **Azimuth**, which is rotation around the horizontal axis.

You define the spot light's cone of light in the *Modify Light Source* window. The cone of light includes the *cutoff angle*, which is the angle from the middle to the edge of the cone, and the *dropoff value*, which measures the amount of light that drops off from the middle to the edge of the cone. As you increase the dropoff value from zero to 128,

more light drops off towards the edge of the cone, and the light beam becomes narrower.



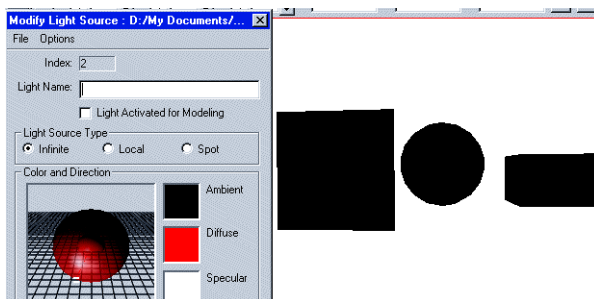
Modeling Light Source

A *modeling* light source is a temporary light source that you use for modeling purposes. You can activate a modeling light to preview lighting effects on your model instead of adding a light source node to the database hierarchy. It does not have a graphical representation in the database but is associated with the eyepoint and moves with the eyepoint. It illuminates everything in the database. A modeling light is saved with the database but is not exported to the runtime system.

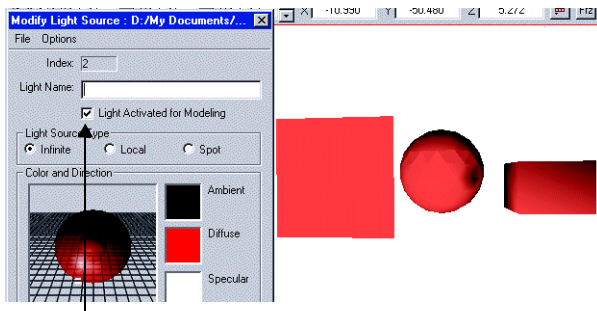
When you first open a Creator database and begin modeling, the light that illuminates your geometry is a modeling light defined as an infinite light. Without the modeling light, your geometry would be dark until you defined a light source node as an infinite light.

To activate a modeling light, you set the **Light Activated for Modeling** checkbox for an existing light source in the *Modify Light Source* window, as shown in the following example. In the **Light Source** palette, choose **Edit/Edit**, or double-click the light source definition to display the *Modify Light Source* window. The default setting for the **Light**

Activated for Modeling checkbox is set until you clear it. A modeling light uses the same properties as its associated light source defined in the **Light Source** palette.



Without a light source placed in the database scene, the objects in the database remain dark



You set the **Light Activated for Modeling** checkbox in the *Modify Light Source* window to define a modeling light source. All objects in the database are illuminated.

This light source counts against the limit of eight light sources that can be rendered in a database. Your runtime system, however, might render more than eight light sources.

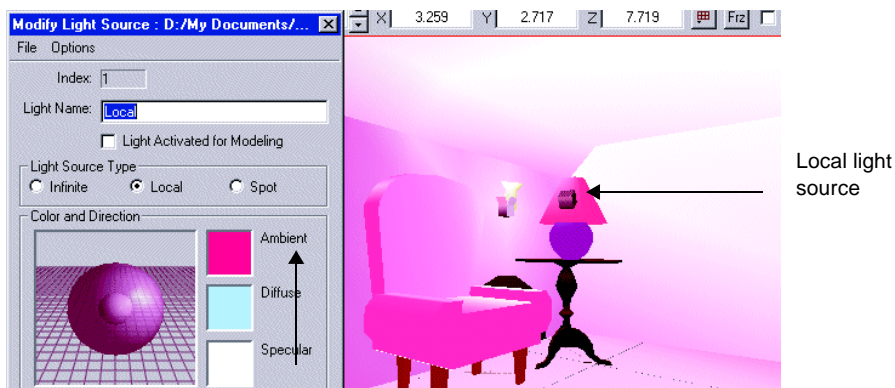
Light Sources and Color

Creator and OpenGL approximate lighting effects using colors. The color qualities associated with a light source (*ambient*, *diffuse*, and *specular*) are combined with the color, texture, and material of an object to simulate lighting effects.

You define a light source's color qualities in the *Modify Light Source* window. In the **Light Source** palette, choose **Edit/Edit** or double-click the light source thumbnail to display the *Modify Light Source* window.

Ambient Color

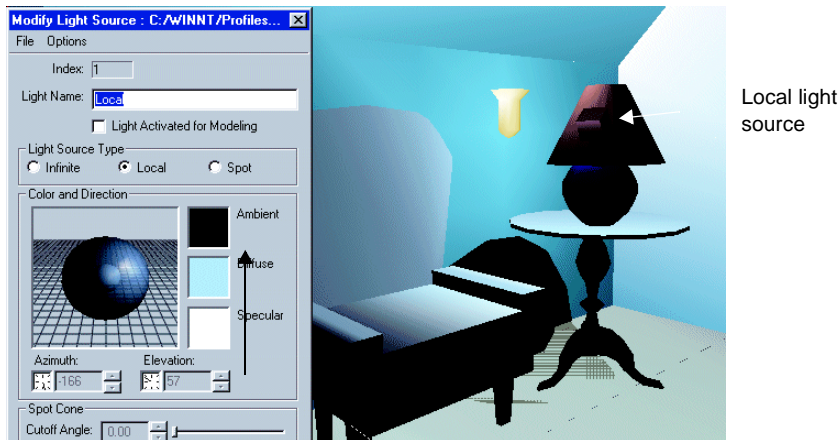
The *ambient* quality is the color of the overall illumination in the scene ("light noise") and is evenly applied on all objects in the scene for infinite and local light sources. For spot light sources, the ambient color shines on objects that lie within the defined cone of effect. The ambient color is most visible on areas facing away from a light source.



The ambient color produced by the local light source on the table lamp affects the entire scene

Diffuse Color

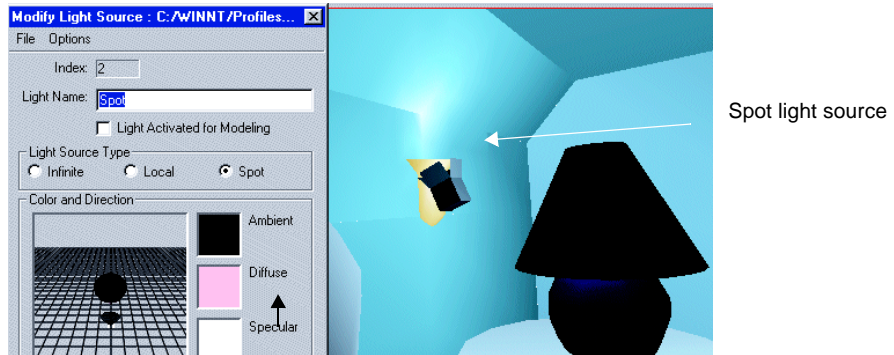
The *diffuse* quality is the color of the light directly from a light source. This color appears on all polygons that surround a light source and is brightest on the polygons that are close and perpendicular to the light source, for example, underneath a lamp.



The diffuse color produced by the local light source on the table lamp appears on polygons that face the light source. The diffuse color is brightest on areas, such as the table and the arm of the chair, that are close and perpendicular to the light.

Specular Color

The *specular quality* is the color of a highlight, which is a bright spot on an object. The specular light color is brightest in the area between the eyepoint and the light's direction vector.



The specular color produced by the spot light source on the wall sconce appears on the wall and ceiling. The specular light is brightest on the polygons that face the direction vector of the light.

Changing Light Intensity

You can adjust the intensities of the diffuse, specular, and ambient qualities for dramatic lighting effects. You can also define *attenuation* effects for local and spot light sources to display in the runtime system. With attenuation defined, light intensity on a model decreases as the model moves farther from the light source. Creator calculates attenuation using values that you enter.

Light Intensity

To adjust light intensity, you can change a light source's location, direction, or color. For example, you can change the size of a highlight by changing the position of a spot light. You can also change the light's color to soften or accentuate lighting.

For the diffuse and specular qualities, you can choose **Attributes/Place Light Source** to change the location and direction of the light source in the *Place Light Source* window. To increase the size of a spot light's highlight on an object, you can move the light source

farther away from the object. To create a more focused highlight, you can move the light source closer to the object.

For the ambient quality, you must adjust the intensities of the diffuse and specular qualities. Because ambient color is applied uniformly to all objects affected by the light source in the scene, the ambient's intensity does not directly change if the location and direction of the light source changes. The ambient's intensity appears to decrease if it is overwhelmed by the diffuse or specular colors, for example.

For all three qualities, you can select the color chip for the diffuse, ambient, or specular color in the *Modify Light Source* window (**Edit/Edit...** in the **Light Source** palette) to adjust the RGB values. For example, you can lower the values of the diffuse color to prevent a “wash-out” of the diffuse color on a face. You usually want to start with an ambient color of either black or beige and then lighten the color for the effect that you want. You can either lighten the specular color or, if you are using a material, adjust the shininess value for the material in the *Modify Materials* window (**Edit/Edit...** in the **Materials** palette) to change the size (intensity) of the specular highlight on an object.

Attenuation

For local and spot light sources, you can define the *attenuation* of the light, which is the decrease in light source intensity as the distance from a local light source increases. Because an infinite light source is not positioned in the database, Creator does not calculate its distance from a model and attenuation of light. You enter values for Creator to calculate either a gradual or a fast decrease in light intensity. Creator displays the effects of attenuation, and the attenuation parameters are passed to the runtime system.

Creator uses values you enter to calculate an *attenuation factor* that controls the light intensity. The attenuation factor defines a mathematical curve to represent reduction in light source intensity with distance. When you define the light source in the *Modify Light Source* window, you enter values (Constant, Linear, Quadratic, Distance) for the attenuation factor. An Intensity@distance value represents the attenuation at a specified distance. If you enter this value and a Distance value, Creator automatically sets the Constant, Linear, and Quadratic amounts for you.

The attenuation factor defines either a linear curve for a gradual decrease in light intensity or a quadratic curve for a faster decrease in light intensity, depending on the values that you enter. You can enter values for Constant, Linear, or both to define a linear curve. You can enter these values and a Quadratic value, or only the Quadratic

value, to define a quadratic curve. The Constant, Linear, and Quadratic values must be between zero and one. The Distance value that you can enter is the maximum distance for the curve, and intensity is reduced along the curve until the Intensity@distance value is reached.

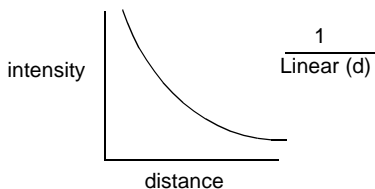
The attenuation factor and the two types of curves are illustrated in the following example.

Attenuation Factor

$$\text{Attenuation factor} = \frac{1}{\text{Constant} + \text{Linear} (d) + \text{Quadratic} (d^2)}$$

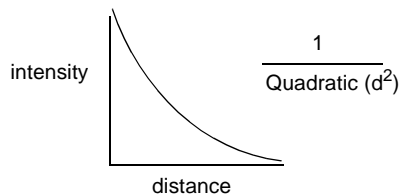
d is the distance from the light source position to a vertex, and *Constant*, *Linear*, and *Quadratic* are the terms that you enter in the *Modify Light Source* window. As d increases, the value of the attenuation factor (light intensity) decreases.

Linear Attenuation



The Linear value is multiplied by the Distance value to produce a linear drop-off in intensity

Quadratic Attenuation



The Quadratic value is multiplied by the square of the Distance value (d^2) to produce a faster drop-off in intensity

Shading an Object

Lighting effects are only visible on an object when the Calculate Shading process is performed, since light is emitted on an object using the direction of the object's vertex normals. Calculate Shading computes the vertex normals for light colors and vertex colors depending on the shading model that you choose in Creator, and shades a face with the computed color. When Calculate Shading is performed (**Attributes/Calculate Shading...**), vertex normals extend from a vertex at a 90 degree angle to the face.

The Calculate Shading process uses light colors, as well as an object's face color, material, and vertex colors to shade the object. You choose one of the following shading models in the *Calculate Shading* window to shade your object according to its properties:

Shading Models

Model	Description
Flat	Only face color is used; no shading is used.
Lit	Face color, material, transparency, and light color are applied; vertex color is not used to shade face. Shading changes dynamically as you add or reposition light sources and objects.
Gouraud	Vertex color is interpolated across the face's assigned color. An assigned vertex color is preserved if you set the Update Vertex Colors option in the <i>Calculate Shading</i> window; otherwise, the vertex color is calculated using face and light colors.
Lit Gouraud	Vertex color is interpolated across the face's assigned color. Material, transparency, and light color are applied. An assigned vertex color is preserved if you set the Update Vertex Colors option in the <i>Calculate Shading</i> window; otherwise, the vertex color is calculated using face and light colors.

Note: You can remove light sources to reduce the runtime's computations after you calculate shading using the Gouraud shading model. With Gouraud shading, the vertex colors are calculated and interpolated across the face. The shading effects remain on the faces after you remove the light sources. This is useful for objects that remain stationary in your scene and do not require changes with lighting effects.

Smooth and Flat Shading

Shading can enhance the structure (curved or angular) of two adjoining faces. For curved surfaces, you want a more gradual change of colors, which is called *smooth shading*. For angular surfaces such as corners, you want a sharper contrast of colors, which is called *flat shading*.

The angle between the vertex normals on the adjoining faces, the light vector, and the *angular tolerance* that you set in the *Calculate Shading* window (**Attributes/Calculate Shading...**) determine the shading effect. The angular tolerance is a limit for the size of

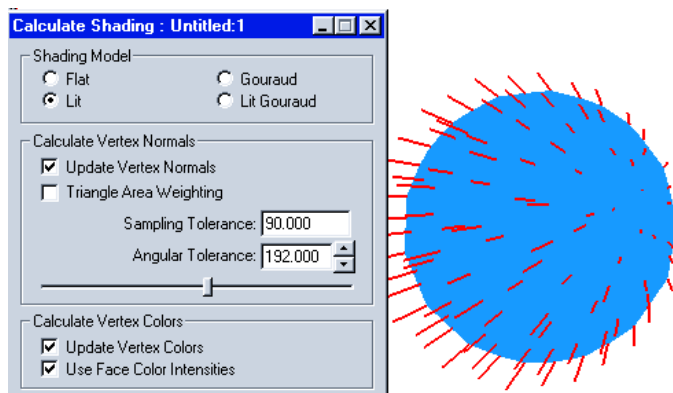
the angle and determines whether the vertex normals on adjoining faces are averaged together (as one normal) or not. Smooth shading occurs when normals with angles less than the tolerance value are averaged together as one normal. Flat shading occurs when the angle between the normals is greater than the tolerance value.

The light vector's direction affects the brightness. Areas where the vertex normals (either averaged or separated) are close to or 180 degrees with the light vector receive the most light. An area is darker when its vertex normal points away from the light vector.

Note: Instead of adjusting the angular tolerance, you can also physically move the normals with the **Modify Vertex** tool in the **Modify Vertex** toolbox to achieve different shading effects. If you change the angle of the vertex normals, freeze the normals in the *Vertex Attributes* window so that they are not repositioned back to 90 degrees if you calculate shading again.

Smooth Shading Example

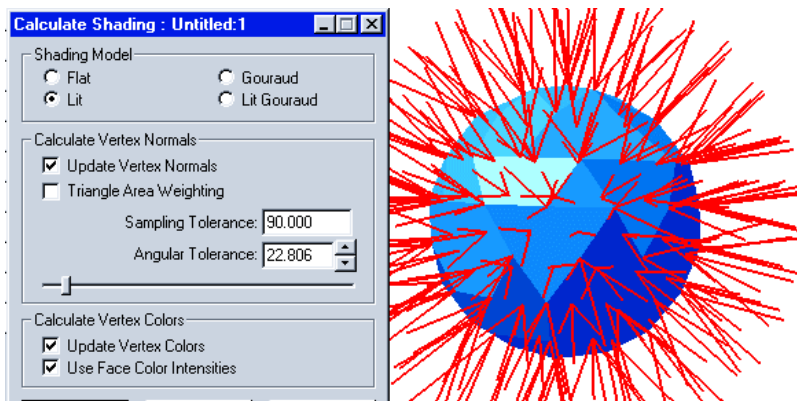
In the following example, smooth shading occurs when the angle between the vertex normals is 90 degrees, and the angular tolerance is set at greater than 90 degrees. Because the angle between the normals is less than the tolerance value, the normals are averaged together. The faces adjoining at the vertex get the same amount of light and color and appear smooth.



The vertex normals on adjoining faces are averaged together when the angular tolerance is greater than the angle between the vertex normals. The faces on the sphere receive the same amount of light for a smooth look.

Flat Shading Example

In this example, flat shading occurs when the angle between the vertex normals is 90 degrees, and the angular tolerance is set at less than 90 degrees in the *Calculate Shading* window. The vertex normals remain separated because the angle between the normals is greater than the tolerance value. Each adjoining face gets its own color and amount of light. Each face is shaded differently from the other, which results in a faceted look.



The vertex normals on the adjoining faces remain separated when the angular tolerance is less than the angle between the vertex normals. The faces on the sphere receive different amounts of light for a faceted look and are a brighter color where the normals face the direction of the light.

Lighting and Materials

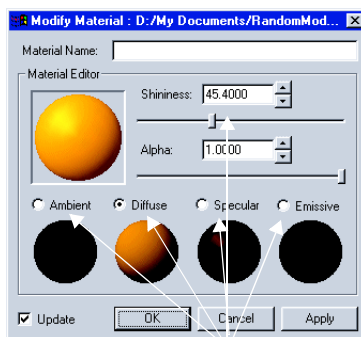
Materials simulate light reflecting characteristics of substances like wood, plastic, and metal. They have their own set of ambient, diffuse, specular, and emissive properties that are often used with light source colors and textures to produce special effects. Emissiveness simulates light radiating from an object for a “glowing” effect. The emissive color is not reflected or affected by the light’s color, as the ambience, diffuseness, and specular colors of the material are.

A material’s specular quality can enhance the shiny aspect of a texture, for example. The size of the specular highlight is controlled with the shininess value that you set for the material. The size of this highlight determines how shiny or dull the object looks. The smaller the highlight, the shinier the object looks. A broad highlight emulates a duller finish.

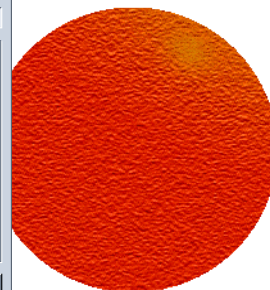
In this model of an orange, an orange material was blended with an orange texture to enhance the color. The ambient, diffuse, specular, and emissive colors for the orange material were defined by clicking on each sphere in the *Modify Material* window (**Edit/Edit...** in the **Material** palette) and choosing a color palette from the *Color Palette* window that appears. The Shininess value that you enter in the *Modify Material* window creates the highlight on the orange.



The orange model with only an orange skin texture applied



The color of the orange is enhanced when an orange material is blended with the orange skin texture. The ambient, diffuse, specular, and emissive colors as well as the Shininess value are all defined for the material.



Materials also have an alpha value for transparency (0 is completely transparent, and 1 is completely opaque) that you define in the *Modify Material* window. If you combine a transparent material with a transparent face, both alpha values for transparency are multiplied together to produce a greater transparency on the object.

Creating Shadows and Other Effects

With Creator tools and a little imagination, you can enhance your lighting effects. You can shade faces to resemble shadows or highlights, for example. You can also brighten colors to resemble lit areas.

Shadows

In the real world, shadows are naturally created on objects when light is emitted on them. Creator does not automatically cast shadows, however. You must create shadow

effects to display in Creator. You can create a shaded face to look like a shadow, if your lights are always going to be on and in the same position in the runtime system.



A face can be constructed with the **Polygon** tool to look like a projected shadow

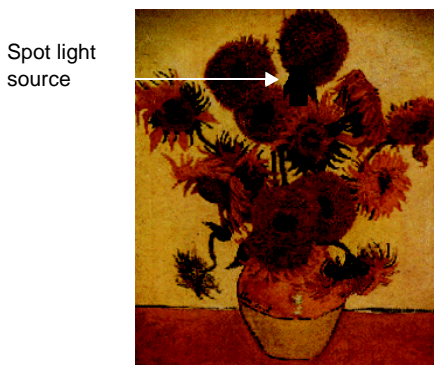
Note: You can set the **Shadow** checkbox in the *Object Attributes* window for your runtime system to generate a shadow for an object.

Bright Light

For finer lighting on your faces, use the **Cookie Cutter** tool to break the face into smaller faces along the grid lines. The **Cookie Cutter** tool cuts an image into squares based on a grid dimension that you choose. You create more polygon vertices for increased lighting.

For example, if you want to enhance a spotlight's effect on a painting, you can use the **Cookie Cutter** tool to slice the image into smaller squares. This will increase the amount

of diffuse and specular lighting that is cast on the image, as shown in the following example.



A spot light source was added in Creator, and the **Cookie Cutter** tool was used to increase the number of vertices and lighting effect on the image

The added squares also increase the polygon count. If you want to preserve your polygon budget, you can increase the brightness of the spotlight's diffuse color in Creator, calculate shading as you normally would, and save and apply the image as a texture.

Spotlights

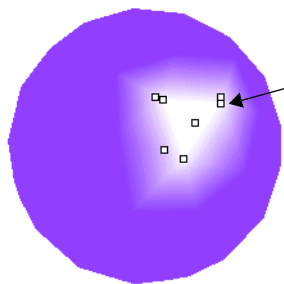
Instead of adding spotlights to your database, you can add spotlight effects to your image in a graphic editing tool, such as Adobe Photoshop, or shade polygons on your model to resemble a highlight.

After you import your image into Adobe Photoshop, you can use a spotlight filter to add lighting to the image. You can apply the image as a texture in Creator. The following example shows the resulting spotlight on a painting.



A spotlight filter for the image was used in Adobe Photoshop to create the spotlight effect

To create a white spotlight on an object instead of adding a spot light source, you can select polygons in a small area to shade. Select the polygons, color the vertices white, choose the Gouraud shading model in the *Calculate Shading* window (**Attributes/Calculate Shading...**) to cast the white vertex color across the face color, clear the **Update Vertex Colors** checkbox so that Calculate Shading does not change the white vertex color, and calculate shading.



These vertices were colored white. Shading was calculated using the Gouraud shading model to create a white spotlight.

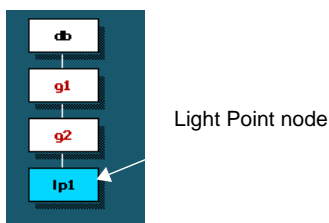
Adding Light Points

You can add light points to simulate small points of light, such as stars in the sky or airport runway lights. Light points are vertices with characteristics and colors that are most visible in a dark sky or fog in your database. Because they are not considered light sources, they do not count against the light source limit of a platform.

To add light points, you use the **Light Point** tool in the **Create** toolbox. In the *Light Point* window that appears, you enter parameters that define the light points' characteristics. After you define how you want the light points to look, you click in the Graphics view to place the light points.

All light points created before you close the tool are placed in a light point node that attaches to the parent group or object node in the database hierarchy. You cannot modify individual light points; if you modify a light point, all light points included in the same light point node also change.

If you move the light point node in the database hierarchy, the light point does not move in the Graphics view. To move a light point, you must use the **Modify Vertex** tool in the **Modify Vertex** toolbox or the **Translate** tool in the **Maneuver** toolbox to translate the vertex to a different location.



Light Point Parameters

The *Light Point* window contains parameters, such as the light point type, direction, display mode, color, and level of detail, that define the light points' characteristics. Creator displays these characteristics, but only the runtime system displays ambient intensity, fading and maximum pixel size, calligraphic properties, and flashing and rotations, which are described in the Creator online help.

Light Point Types

The light point type in the *Light Point* window specifies the manner in which light points are distributed along a line, curve, grid, or within a defined area. For the basic Creator application, the setting is fixed at Light Points, which places a light point at each coordinate that you click with the mouse.

If you have the CreatorPro option, which enables you to create terrain databases, you can also create light strings, light grids, and random light points. You can also automatically create light points when you generate terrain. The Creator online help includes descriptions of all the light point types.

Viewing Direction

The direction parameter specifies the display of light point color based on the viewing direction. When you choose **Attributes/Directionality** from the **Attributes to Modify** list in the *Light Point* window, viewing directions are available in the **Light Point Directionality** group.

These are the viewing directions that you can choose from:

- **Omnidirectional** - Visible from all sides
- **Unidirectional** - Visible from only one side
- **Bidirectional** - Visible from only the front and back

Omnidirectional Lights

Omnidirectional lights display one color from all viewing directions. The color that displays is the front-facing color, which is the color that you choose on the Frontfacing Color Band in the *Light Point* window. This type of light point is useful for runway edge lights, which are visible from all directions, as shown below.

Runway edge lights

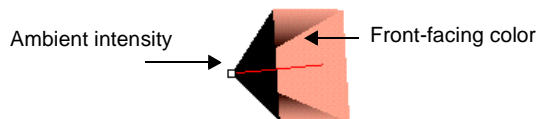


Unidirectional Lights

Unidirectional lights display a front-facing color on the positive side of a light point's direction vector and ambient intensity on the negative side. You define the i, j, k values for the light point lobe's direction vector in the *Light Point* window.

The ambient intensity that you set (between 0 and 1) is the brightness value that falls outside of the light point lobe. Lobes define the field of visibility for the light. An ambient intensity of 0 indicates that the light is completely off and 1 indicates the light is at full intensity outside of the light point lobe. The ambient intensity does not display in Creator, however, but displays its intensity value in the runtime.

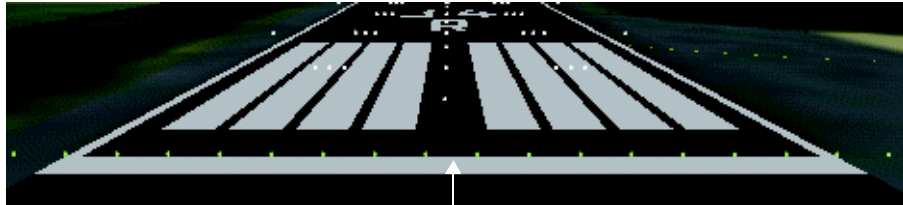
The light point lobe is an angular cone that represents unidirectional and bidirectional light points. It is easier to set the light point's direction with the lobe normal and angles at which a light point is visible if you display the light point lobe. See "Light Point Lobes" on page 4-51 for more information about how light point lobes are defined and used.



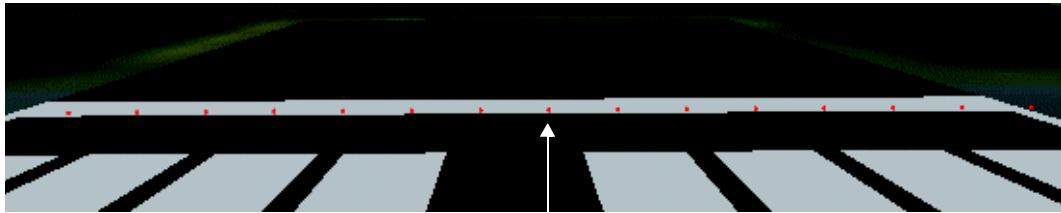
Bidirectional Lights

Bidirectional lights display a front-facing color on the positive side of a light point's direction vector, a back color on the negative side, and ambient intensity outside the light point lobes. You define the i, j, k values for the light point lobe's direction vector in the *Light Point* window.

Bidirectional lights are useful as runway end lights, which are green (front-facing color) on the approach side and red (back color) on the opposite side, as shown in this example.



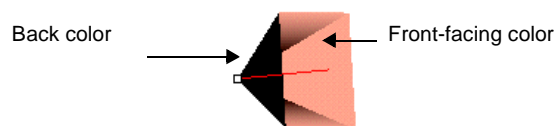
Runway end lights are green on the approach side (front face)



Runway end lights are red on the opposite side (back face)

The ambient intensity that you set (between 0 and 1) is the brightness value that falls outside of the light point lobe. Lobes define the field of visibility for the light. An ambient intensity of 0 indicates that the light is completely off and 1 indicates the light is at full intensity outside of the light point lobe. The ambient intensity displays as invisible in Creator, but displays its intensity value in the runtime.

The light point lobe is an angular cone that represents unidirectional and bidirectional light points. It is easier to set the light point's direction with the lobe normal and angles at which a light point is visible if you display the light point lobe. See "Light Point Lobes" on page 4-51 for more information about how lobes are defined and used.



Display Mode

Display mode specifies the method used to display light points on the computer screen. You can choose to display light points in either *raster mode* or *calligraphic mode* depending on your hardware. When you choose **Attributes/Directionality** from the **Attributes to Modify** list in the *Light Point* window, display modes are available in the **Light String Attributes** group. Creator only displays light points in raster mode.

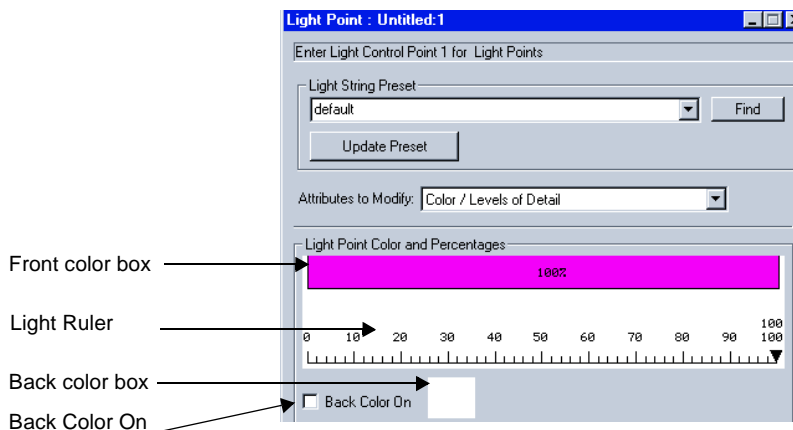
The computer screen usually displays rasterized images, where pixels are displayed as they are drawn in lines across the screen. Rasterized light points, however, appear to be “blocky” and somewhat dull.

Calligraphic light points are bright and well-defined. Calligraphic images require special hardware for plotting characters using an analog beam of light.

Color

Color parameters assign the front-facing colors and back-facing color to light points. When you choose **Color/Level of Detail** from the **Attributes to Modify** list in the *Light Point* window, light point color parameters are available.

To select a front-facing color, double-click the **Front** color box to open the **Color** palette, as shown in the following example. Select a primary color and click the **Front** color box once. To select the back-facing color, double-click the **Back** color box to open the **Color** palette. Select a new primary color and click the **Back** color box once. You can set the **Back Color On** checkbox to use the back color instead of the front color, which can only be viewed in the runtime system.



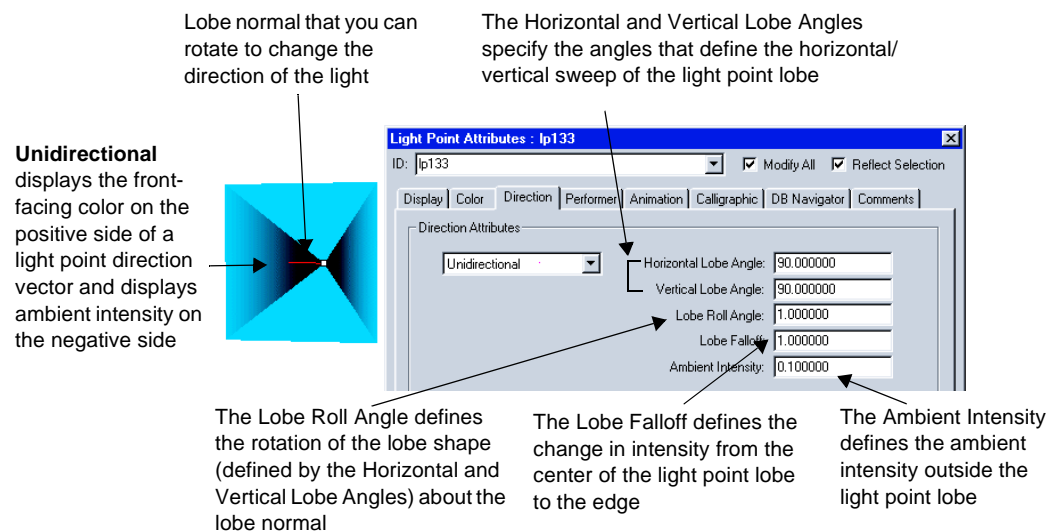
If you have the CreatorPro option, you can use the Light Ruler to set the number of front-facing colors and the distribution of color (as a percentage) for a random distribution of light points. You can define up to eight colors. See the Creator online help for directions on using the Light Ruler.

Light Point Lobes

The light point lobe is an angular cone that represents unidirectional and bidirectional light points. It is easier to set the light point's direction with the lobe normal and angles at which a light point is visible if you display the light point lobe. To see a light point's lobe, you set **Draw Vertex Normals** in the **View** panel because light points are vertices.

With the light point lobe displayed, you can rotate the lobe normal using the **Modify Vertex Normals** tool in the **Modify Vertex** toolbox. This is the same as defining the light point's direction vector (lobe normal) with i, j, k coordinates in the *Light Point* window.

You can define the angles at which a light point is visible with the lobe parameters in either the *Light Point* window or the *Light Point Attributes* window (**Attributes/Modify Attributes**). The lobe parameters include the horizontal and vertical angles, the lobe roll angle about the vertex normal, and the lobe falloff (change of intensity) value from the light lobe center to its edge. The parameters in the *Light Point Attributes* window are shown in the following example.



Defining Light Points for LODs

As you zoom out from a scene with the mouse, you want the light points to gradually fade away instead of staying at the same intensity. If all of the light points remain in the scene, they remain fixed in size and appear to cluster together as you zoom out. You can assign light points to levels of detail (LOD) in the *Light Point* window so that they turn off when they are viewed at specified distances. LODs are versions of the same geometry with different numbers of polygons.

For an LOD, a percentage of light points are retained from the next higher LOD. The LOD retains this percentage and randomly removes the rest of the lights. In this way, you can create the illusion of random lights gradually fading away from your view as you move farther from the scene.

The LOD scale is available when you choose **Color/Level of Detail** from the **Attributes to Modify** list in the *Light Point* window. In the LOD scale, you set the number of LODs and the switching distance of each LOD. See the Creator online help for directions on using the LOD scale.

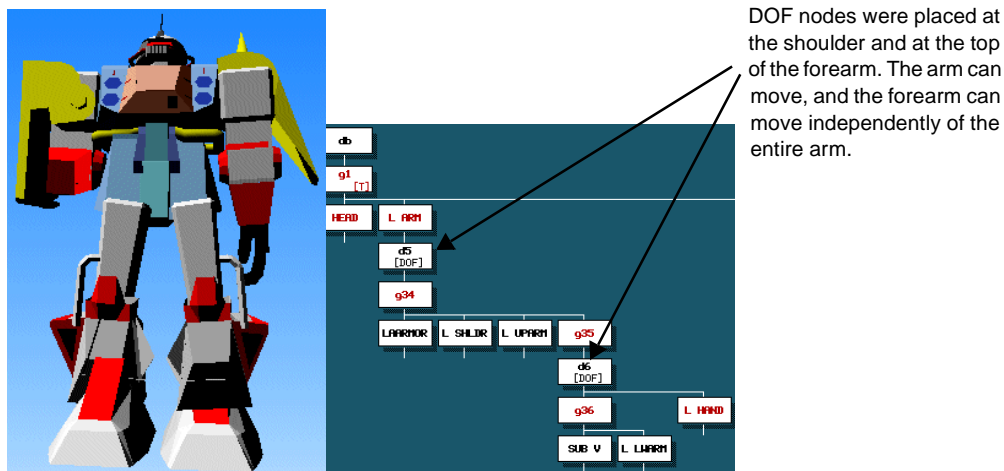
See “Using Levels of Detail” on page 6-17 for more information about LODs.

Adding Movement

If you want specific parts of your model to move, you can add Degree of Freedom (DOF) nodes to those areas. A DOF establishes a *local coordinate system*, and the geometry it controls moves around the axes of the coordinate system. A local coordinate system is a coordinate system that the user defines for modeling away from the database origin. A DOF node contains variables for rotation, scale, and motion, so that you can specify a full range-of-motion in three dimensions.

The range-of-motion applies to all descendants of the DOF node in the database hierarchy. This ensures that all elements of that DOF node are equally affected. For example, when a robot’s arm moves, as shown in the following example, you also expect its hand to move along with it. After you create a DOF node, you would place the DOF node on top of the arm’s node in the hierarchy. If you wanted the wrist below

the hand to move independently of the arm, you would add an additional DOF node above the hand's node.



First create a DOF node using the **Create DOF** tool in the **Create** toolbox and attach it to geometry in the database hierarchy to which you want to add movement. Then, complete these remaining tasks:

- Create a local coordinate system for the DOF.
- Define movement along the axes using limits that reference the local coordinate system.
- Check the range of movement to see if you are satisfied with it. You can change the limits to change the motion.

Creating a DOF Local Coordinate System

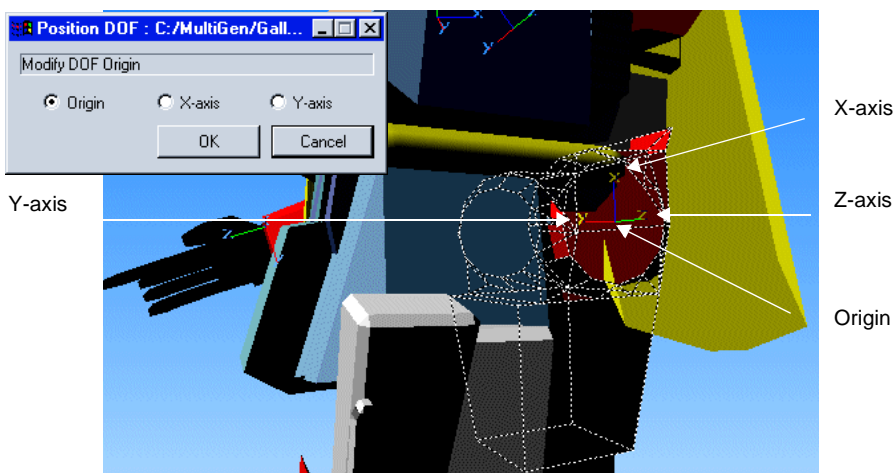
Movement for a DOF occurs around the x , y , and z -axes of a local coordinate system. You can define a local coordinate system with its own origin when you model at distances away from the global axis. You orient the x , y , and z -axes on the part of a model that you want to move.

When you create a DOF node using the **Create DOF** tool in the **Create** toolbox, the axes of the local coordinate system are automatically positioned at the database origin (0,0,0). You must position the axes on the part of the model attached to the DOF using

the *Position DOF* dialog box. See “Defining a Local Coordinate System” on page 5-1 for more information about local coordinate systems.

Note: **Draw DOF Axes** must be active in the **View** panel to see the axes.

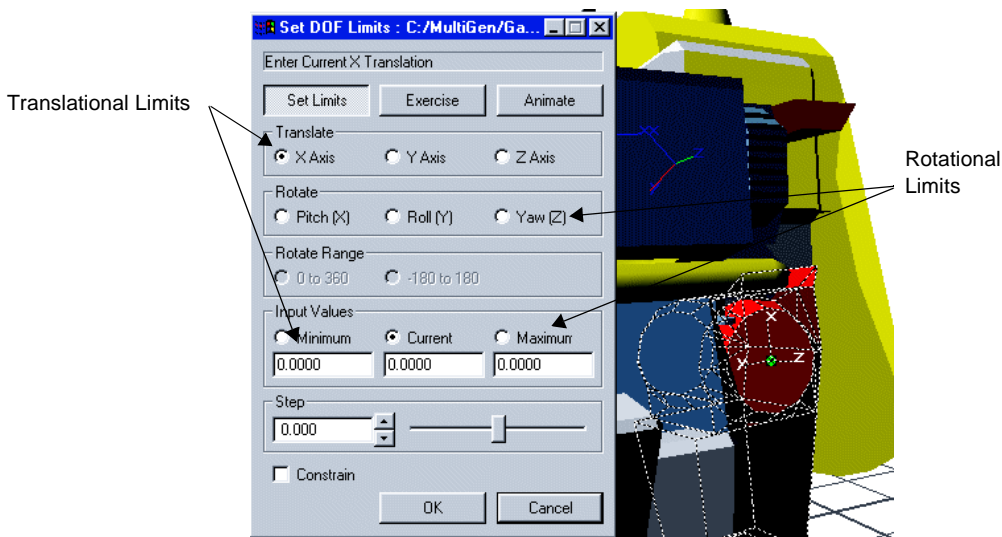
To position the DOF axes, you choose an origin, x-axis alignment point, and y-axis alignment point. The locations you choose for the origin and alignment points depend on the movement you want for your model (up, down, or sideways). On the robot model in the following example, the local coordinate system was placed at the arm’s elbow to move the forearm around the z-axis.



Defining Movement

You assign movement to the geometry attached to the DOF by setting DOF limits in the *Set DOF Limits* dialog box, as shown in the following example. You define *translational limits* that specify the distance that a DOF can move relative to its local coordinate system, *rotational limits* that specify a range in degrees from 0 to 360 through which the DOF can rotate around each axis, as well as *pitch* (rotation around the x-axis), *roll*

(rotation around the y-axis), and yaw (rotation around the z-axis). By setting the limits, you can specify a full range of motion in three dimensions.



Checking DOF Movement

After you have positioned the DOF axes and defined the movement, you are ready to check to see if the movement is what you have intended. You can either manually move the DOF by choosing **Exercise** in the *Set DOF Limits* dialog box, or automatically move the DOF according to the DOF limits by choosing **Animate** in the *Set DOF Limits* dialog box.

For a lesson on adding movement to a model using DOFs, see “Degrees of Freedom” in the *Desktop Tutor*.

Adding Sound

You can add sound to a model by assigning a sound file to a sound node in the database hierarchy. You place the sound node in the database from where you want to emit the sound, for example, in an airplane’s engine. In Creator, you create the sound node to play during runtime. Depending on how you program your runtime system, the sound plays when the model is in the field of view and a certain distance from the eyepoint.

When you create the sound node in Creator, you can define many of the sound attributes to control the physical properties of the sound, such as the amplitude (volume), and falloff (the rate at which the amplitude falls off). With Windows and a sound card installed, you can play *.wav*, *.avi*, *.mid*, and *.rmi* files in Creator using the Windows default sound player utility *mplay32.exe*. For IRIX systems, the default utility is the SGI Sound Editor.

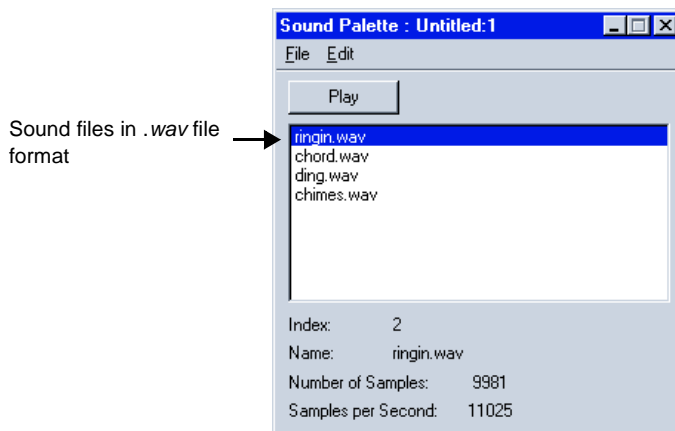
To add sound in the database, complete the following tasks:

- Load a sound file into the **Sound** palette
- Create a sound node
- Assign a sound file to a sound node
- Assign a coordinate position to the sound node

Loading Sounds

You load a sound file into the **Sound** palette file, for example, *sound.sp1*, that contains a list of the sound files to use in the database. This **Sound** palette file is created the first time that you save a **Sound** palette with a new database.

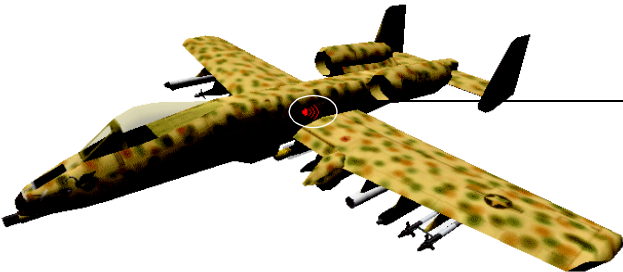
The **Sound** palette file (in *.sp1* file format) is a text file that contains the index, directory path, and file name of each sound file loaded into the **Sound** palette, as shown in the following illustration. You can save all sound files in either the same *.sp1* file or in separate *.sp1* files to load into your database.



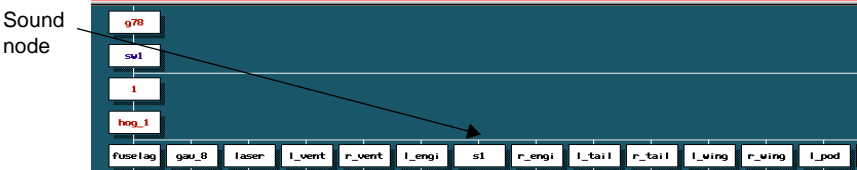
Creating a Sound Node

You click **Create Sound** in the **Create** toolbox to create a sound node. After you select a sound file in the **Sound** palette, the selected sound file is automatically assigned to the node. You can load a different sound by selecting a different sound in the **Sound** palette or by opening a different **Sound** palette file and selecting a sound file.

Sound nodes function as group nodes in the database hierarchy. They are attached to groups and DOFs, for example, and can have any node type as a child.



A small, sound icon appears at the database origin when you create a sound node. You can move the sound icon to any location within the Graphics view.



When you create a sound node, a small sound icon appears in the Graphics view at the database origin. You can position the sound icon anywhere in the Graphics view using the *Position Sound* window; however, the sound node in the database hierarchy does not move. Before you create additional sound nodes, be sure to move the first sound icon away from the database origin. Otherwise, the sound icons are layered directly on top of each other.

5 Exploring Methods to Simplify Modeling

Besides basic modeling tools, Creator provides tools that help save time and effort when you create models. You can use these techniques as you develop your database scene, or use them to modify and enhance your models. You can also advance your modeling skills when you learn easier methods for constructing elements of your database. These methods include:

- Using *local coordinate systems* when you model away from the database origin. With a local coordinate system, you can define an origin wherever you want it to keep the coordinate size small. For example, after you define a local origin, you can work with coordinates such as (5, 3) instead of (1200, 1360).
- Using *construction edges* to help you draw straight edges of a polygon and *construction curves* to help you draw curved edges of a polygon. Construction edges and curves are temporary points of reference that are not saved with the database.
- Using *background images* as backdrops while you create models. After you load a 2D image into a database, you can accurately create a 3D model with the same dimensions as the image.
- Building databases with *external references* and *instances*. With both external references and instances, you can reference a model instead of copying and pasting it in a database to reduce the size of the database file. An external reference is a reference to a model in another database. An instance is a reference to a model within the same database.

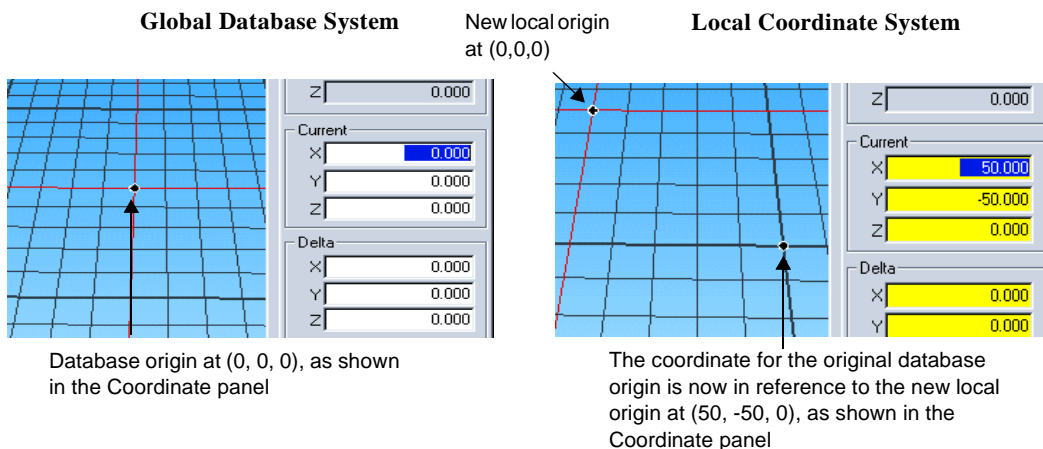
Defining a Local Coordinate System

You can define a coordinate system in addition to the global database system called a *local coordinate system*. A local coordinate system has an origin wherever you want it, for example, in the middle of a model or at a specific location on the tracking plane. You can use a local coordinate system to simplify the coordinate system when working with geometry far from the database origin. The local coordinate system is used for modeling purposes that is not saved in the database.

A local coordinate system is useful when you want to position and work with models in large distances away from the database origin. For example, instead of moving an

object's vertex from (2875, 1046) to a different coordinate, you can define a local coordinate system for the object to simplify the coordinate system in a local area of your database. A large coordinate such as (2875, 1046) would change to a much smaller coordinate relative to the local origin.

You select a coordinate for the local origin in the *Set Local* dialog box. The local origin becomes the new database origin. All other vertices that you model are in reference to this new origin. As you can see in the following example, the original database origin (0, 0, 0) in the global database system becomes (50, -50, 0) after the local origin is created.



Defining a Local Coordinate System With a Transformation

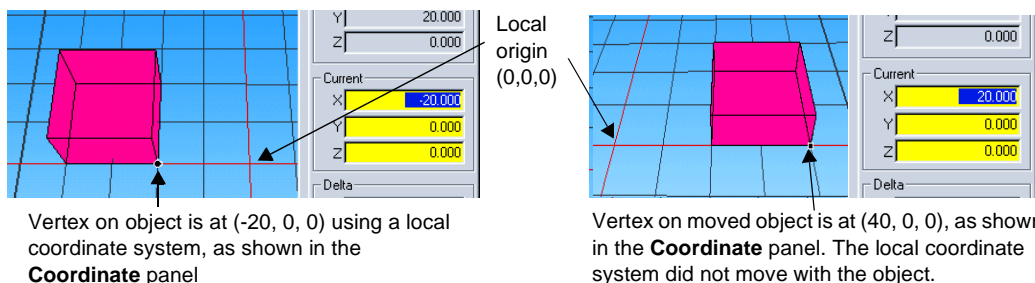
A local coordinate system is useful for adding or refining geometry on a model that has been displaced from the original database coordinates with a *transformation matrix*. A transformation matrix is attached to a node in the database hierarchy to modify the position of several objects at once instead of modifying the coordinates of each individual object. When geometry is modified, the values in the matrix are multiplied against the geometry's original coordinates to produce new coordinates.

To add the local coordinate system, you select a node below the node with the transformation in the database hierarchy and choose **Local-DOF/Set Local From Path**. The local coordinate system is centered on the object in the Graphics view.

See “Applying a Transformation Edit” on page 5-12 for more information about transformation matrices.

Moving an Object

If you define a local coordinate system and then translate or move an object, the local coordinate system is not moved with the object. You can see in the **Coordinate** panel that the new local coordinates are changed relative to the object’s original position, as shown in the following example. You must define a new local coordinate system for your translated object to update its local origin and other coordinates.



Creating Construction Edges and Curves

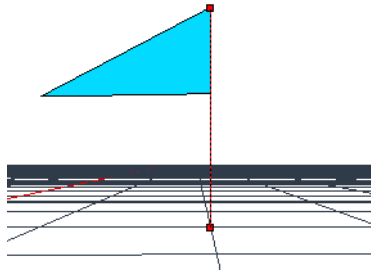
Construction edges and construction curves are similar to stencils for creating shapes. For example, you can create a construction edge and add vertices on it to help you draw a straight edge of a polygon. In a similar manner, you can use the **Construction Curve** tool to create a curved edge of a polygon. Construction edges and curves are temporary points of reference that are not saved with the database.

Construction Edges

Creator provides many **Construction Edge** tools, located in the **Edge** toolbox, for placing construction edges in different areas of geometry. For example, you can use the **Edge From Mouse** tool to create a construction edge from a pair of vertices that you enter with the mouse, or the **Parallel to Edge** tool to create a construction edge parallel to an existing edge in the database. Other types of **Construction Edge** tools that you can use

to create edges are the **Perpendicular to Tracking Plane** tool for creating construction edges perpendicular to the tracking plane, **Centerline** tool for creating a construction edge that joins the midpoints of selected edges on a face, and **Intersection of Two Planes** tool for creating a construction edge at the intersection of two planes.

The **Construction Edge** tools are located in the **Edge** toolbox. After you choose a **Construction Edge** tool, you can click the middle-mouse button to add vertices of a polygon along the edge. See the Creator online help for detailed steps for using the **Construction Edge** tools.



The **Perpendicular to Trackplane** tool was used to create a construction edge perpendicular to the tracking plane. The polygon was created by middle-clicking to add vertices along the construction edge.

Construction Curves

The **Construction Curve** tool creates a connected sequence of curved construction edges (segments). You can use the **Construction Curve** tool to draw curved lines and curved edges of a polygon.

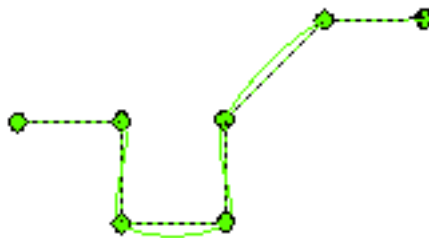
You control the shape of the curve with vertices called *control points*. A curve with these control points is known as a *spline*, which dates back to when shipbuilders used wooden planks to build ships. To bend a wooden plank, shipbuilders would place it between a series of fixed posts. The resulting bent plank became the spline. To reshape the spline, one or more of the posts were moved. In a similar manner, you move control points to reshape curves.

The curved line can either pass directly through the points or near the points. There are two categories of splines: *interpolating splines* and *approximating splines*, as described in the following sections.

Interpolating Splines

The interpolating spline passes directly through each of the control points. An advantage of interpolating splines is the direct relationship between placement of points and direction of the curve. You know that the curve will follow wherever you place a point, since the line passes through the point. A disadvantage, however, is that a perfectly smooth curve is difficult to construct. If a point is not correctly aligned with the other points on the curve, a bump can form.

The type of interpolating spline that Creator supports is called the *Cardinal spline*, in which the curve passes directly through all but the first and last control points:



Cardinal Spline

Approximating Splines

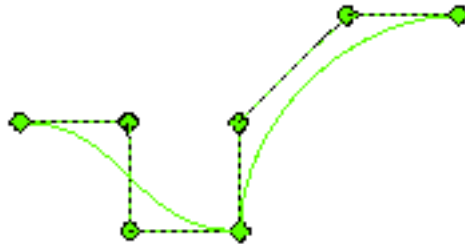
An approximating spline is more gradual than an interpolating spline. The curve line passes near, instead of through, the control points. You have a much wider margin of error when placing the control points, resulting in a much smoother curve.

Creator supports two types of approximating splines: the *B spline* and the *Bezier spline*. The B spline is similar to the Cardinal spline, but the curve passes near instead of through all except the first and last control points:



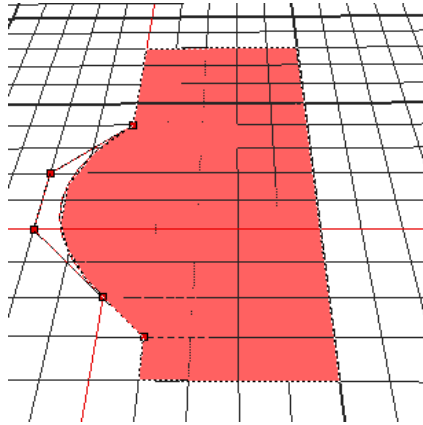
B Spline

The Bezier spline passes through every third control point, including the first and last control points, and passes near all other control points:



Bezier Spline

In the following example, a Bezier spline was used to create a smooth, curved edge of a polygon. After the construction curve was created, vertices were added using the middle-mouse button to create the curve.

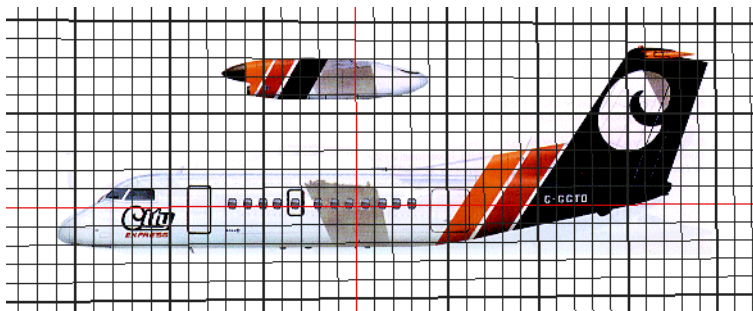


The **Construction Curve** tool was used to create a Bezier spline. The curve on the polygon was created by middle-clicking to add vertices along the construction curve.

Using Background Images

You can either scan or load an image into Creator to use as a background image in the database. With an image in the background, you can accurately create a 3D model that has the same dimensions as the image. For example, you can create a polygon by tracing an outline on the image with the mouse. You can digitize a drawing with the mouse by converting a blueprint of an object into a model that is true to scale.

When you open the image in the Graphics view, the image is set behind the tracking plane. An image that you load into a Creator database can be in any of these file formats: RGB, RGBA, INT, INTA, TIF, GIF, JPG, PCX, LBM, TGA, PIX, or BMP. You can move the image anywhere in the Graphics view as well as resize it.



This image was loaded as a background image. Faces can be created on the tracking plane with the background image as a reference.

You can align geometry with an image by choosing three points on a model and two points on the image. Creator adjusts the eyepoint to match the first two points on the model with the two image points. The third point on the model defines the plane the eyepoint looks toward. Instead of scaling and rotating the database to match an image, Creator aligns geometry with an image by adjusting the eyepoint, not changing the database. You can save the eyepoint position and tracking plane position so you can create other geometry without changing the original alignment.

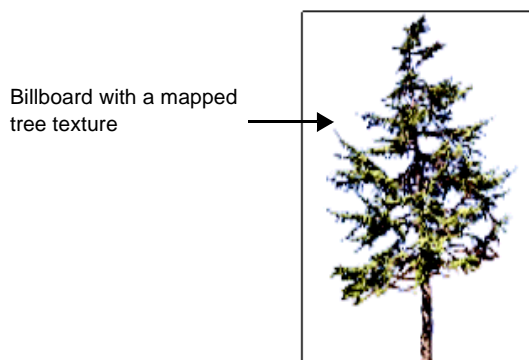
Creating Billboards

Billboards are single polygons that are often used to represent symmetrical objects, such as poles, trees, or people. You can apply texture to billboards to create very detailed objects with a low polygon count. Billboards rotate to face the eyepoint in the runtime system, so the textured side of the polygon is always visible.

To create a billboard, you model a polygon at the database origin, since billboards rotate around the database origin within the x - z or y - z coordinate plane. If you want to move the polygon to a different location, you must apply a transformation edit to its parent node and translate the polygon. The transformation matrix above the polygon transforms the location of the polygon's local origin and z -axis. If you do not apply a transformation edit to the billboard, the billboard continues to rotate around the database origin. See "Applying a Transformation Edit" on page 5-12 for more information about transformation edits and transformation matrices.

The billboard revolves around either the z -axis or a point to continually face the eyepoint. After you create a billboard, you open the polygon's *Face Attributes* window to choose Billboard options in the **Drawing** panel. You can choose **Axis With Alpha** to rotate the billboard around the z -axis or **Point With Alpha** to rotate the billboard around the billboard's local origin.

You can also instance a billboard and apply a transformation to make it easy to use more than one copy of the billboard in the database. An *instance* is a reference to an existing geometry in the database. Trees are often created using billboards and then instanced to create duplicate copies throughout the database. See "Using Instances" on page 5-11 for more information about creating instances.



Using External References

An *external reference* is a reference to geometry in another database. External references streamline the drawing and editing processes of large databases because contents in another database file are referenced instead of saved in the database. With external references, you can reduce the number of polygons in a large database. The directory path and file name for the other database file are saved in an external reference node.

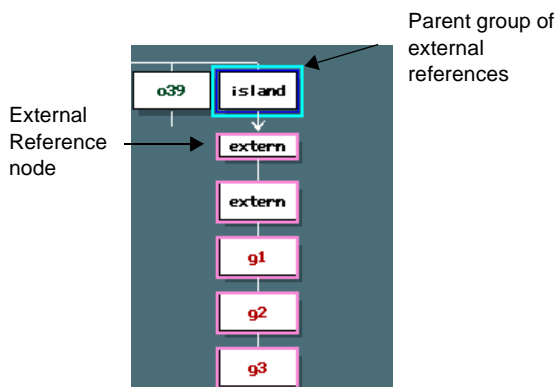
External references are useful for large or complicated databases that use the same geometry in different areas. For example, if a house is in different levels or “worlds” of a video game, the house can reside in the database that forms the first level and can be present as an external reference in all other levels. If you make any changes to the house in the database, the changes are automatically made to all other representations of the same house.

To create an external reference, you place an external reference node in the database hierarchy, and then assign the directory path and file name of another database to the node. You add an external file’s directory path and file name in the *X-Ref Attributes* window when you select the external reference node. Creator uses this directory path to load each external reference.

The referenced geometry is visible in the Graphics view of the target database if the **Read External References** preference is set in the **Flight** panel of the *Preferences* window. You can select individual external reference nodes to display by selecting them in the hierarchy and choosing **File/Reread Externals**.

After you create the external reference node, you position the external reference in the Graphics view by selecting origin, alignment, and third points in the target database. You can also select points in the externally-referenced database that map to points in the target database.

You can edit the geometry only in the externally-referenced database. You must open the external file separately to edit. The changes are made at every location that uses the external reference.



Using Instances

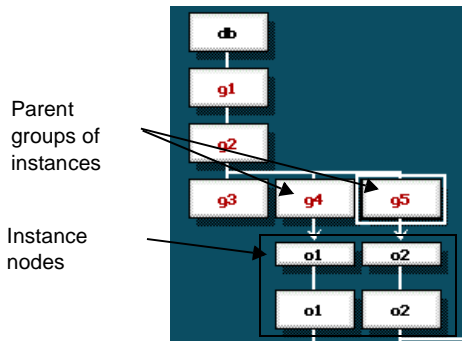
An *instance* is a reference to geometry in the same database. Instances help save disk space and memory because geometry in your database file is referenced instead of duplicated. Only one copy of an object's geometry is stored in memory. If you select or edit one instance to make changes, all other instances are simultaneously selected and changed.

Instances are useful for common objects that you want replicated throughout your scene. For example, if you need to populate your scene with trees, it is a good idea to have an original tree and reference it for all other duplicates of the tree. You can have multiple instances for different types of trees as well.

The advantages of using instances are that you can save disk space, save time creating new models, and save time editing all occurrences of a model. A possible disadvantage is that you have multiple objects that look exactly the same. This may not be noticeable in large, complicated databases but may be a disadvantage in smaller databases in which you need more flexibility.

To create a new instance node, select a node to be instanced and then use the **Create Instance** tool in the **Create** toolbox. The instance node is automatically placed on top of the original geometry's node. In the Graphics view, the instance is created at the same

coordinates as the original geometry, so it is not immediately visible. You must relocate the instance in the Graphics view using a tool in the **Maneuver** toolbox, such as the **Translate** tool, to see the geometry.

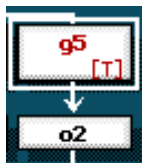


Applying a Transformation Edit

A transformation, such as a translation or scaling, is a convenient way to modify the position of several objects at once without individually modifying each of their coordinates. You apply one or more transformations to a node and all of its descendants in the hierarchy using the **Insert Transformation Matrix** tool in the **Maneuver Tools** toolbox. When you apply a transformation to a node, you attach a *transformation matrix* to the node. A *transformation matrix* is a 4 x 4 matrix of coordinates used to calculate new positions or orientations of geometry. When geometry is modified, the values in the matrix are multiplied against the geometry's original coordinates to produce the new coordinates.

If a node has a transformation matrix attached to it, the node is shown with a letter that indicates the type of transformation, such as **T** for **Translate**, as shown in the following example. Other types of transformations include scaling geometry, rotating geometry about edges, rotating geometry about points, and putting geometry, which

simultaneously translates, rotates, and scales geometry. You can apply more than one transformation to a node.



Transformation matrix attached to a node. The **T** on the node indicates a **Translate** transformation.

If you apply a transformation to an external reference or an instance, the transformation matrix must be loaded and processed as the runtime system accesses or culls each reference or instance. Runtime performance is slowed when the image generator loads each transformation matrix, so it is generally a good idea to use as few as possible.

Direct geometry, in which the coordinates are already in world space, is drawn faster than geometry with a transformation matrix. In a smaller database, direct geometry may be practical. In a large database, it may be more important to save disk space and reduce rendering speed using external references.

6 *Optimizing for Performance*

As you interact with your realtime application, the application needs to be updated at regular intervals to maintain smooth continuous movement. The time delay from user input until application output response, called the *latency period*, should not be detectable when you run the simulation. You can use these modeling techniques as you create an OpenFlight database to increase the smoothness and speed of Creator and your realtime application:

- Organize the nodes in your database hierarchy according to how the runtime system processes data and the size of the database. The organization of nodes affects how efficiently the runtime system traverses the hierarchy to cull and draw data.
- Assign *bounding volumes* to group nodes. A bounding volume is an invisible wireframe shape, such as a box or sphere, that surrounds a model. The runtime system checks if the bounding volumes for group nodes intersect the viewing volume. Nodes outside of the viewing volume are culled and not displayed.
- Reduce the number of polygons. If your runtime system processes too many polygons, frame processing might not be completed in time to transfer into screen memory. You can use techniques to reduce the number of polygons in a database and still maintain the desired amount of detail.
- Adjust *clipping planes* to control the objects in the Graphics view that are drawn. Clipping planes define the near (inner) and far (outer) limits of the viewing volume. The viewing volume is the portion of the database that is visible in the Graphics view. You can increase speed in Creator by adjusting the viewing volume to show only part of the database.

Structuring the Hierarchy for Efficiency

It is important to organize the nodes in the database hierarchy according to how your runtime system processes the data for culling and drawing purposes. Although there are many ways in which you can organize the nodes, the hierarchy structures described in this section can help you maximize performance.

The Cull Process

During the *cull process*, the runtime system continuously traverses the database hierarchy nodes to search for geometry that is in view as the eyepoint changes. Group nodes are checked to see if their *bounding volumes* intersect the *viewing volume*. Bounding volumes are invisible shapes, such as a box or sphere, that enclose a node's geometry so that the runtime system can estimate the geometry's size. The viewing volume is the portion of the database that is currently visible. The group nodes are only traversed if their bounding volumes intersect the viewing volume. Nodes outside of the viewing volume are culled and not displayed.

It is important to organize your database nodes so that the runtime system can efficiently cull and traverse the database hierarchy. You can organize the database in one of three ways:

- **Linear** - All object nodes are arranged under a single group node. The runtime must check every node to see if its geometry is in view.
- **Logical** - Object nodes are separated and grouped under their logical group node. For example, if trees and buildings are object nodes, all trees are placed under one group node and all buildings are placed under another group node. The runtime system must traverse every object node to find the geometry to display.
- **Spatial** - Nodes are organized and grouped by their physical location in the database scene. The runtime culls at the top group level. Group nodes are only traversed if they are in view, which reduces culling time. Most runtime systems cull fastest with a spatially organized hierarchy.

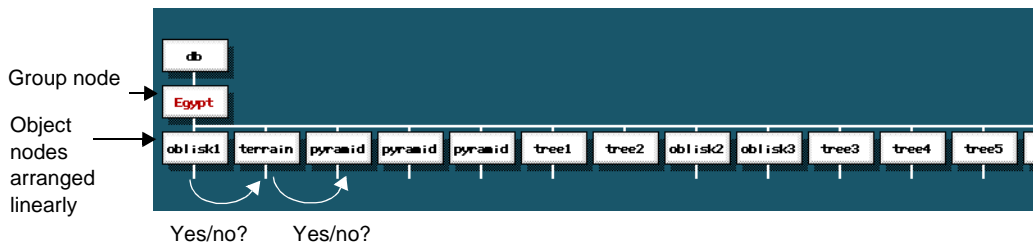
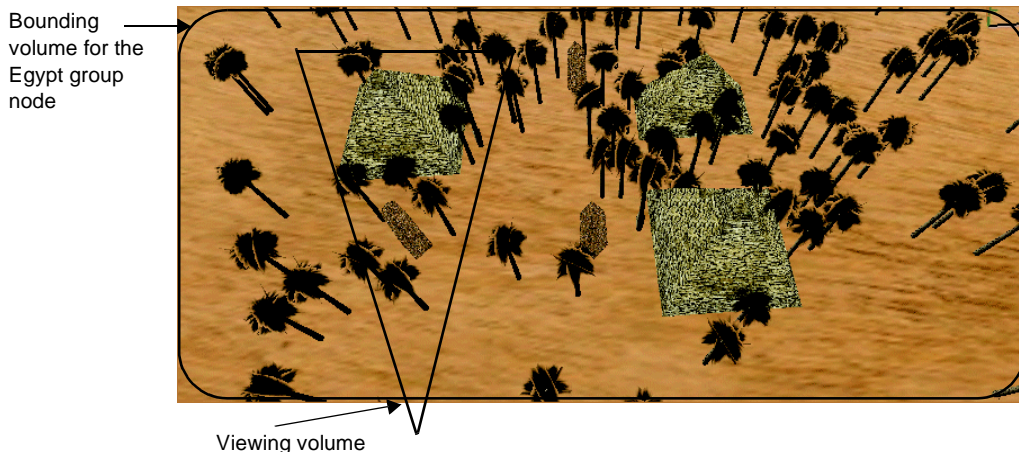
To illustrate these database arrangements, the same visual scene of pyramids, obelisks, and trees is arranged with linear, logical, and spatial hierarchies to compare their effects on culling performance. Pyramids, obelisks, and trees are object nodes placed under an Egypt group node.



Linear Structure

In the following example, the database hierarchy is organized in a linear structure, in which all object nodes are placed side-by-side under the Egypt group node. Because all of the geometry in the database is within the group node's bounding volume, the image

generator must check each object node in the hierarchy to see if it is in the viewing volume to display. Cull performance slows down when each object node is checked.

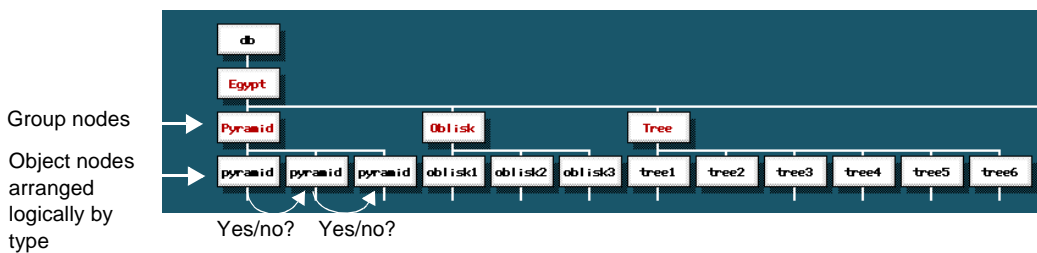
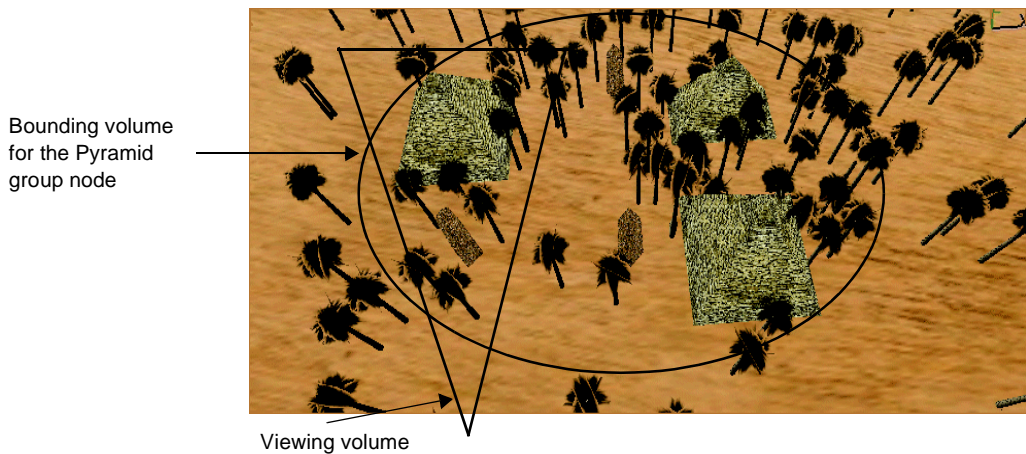


The image generator checks each object node to see if it is in the viewing volume

Logical Structure

In this arrangement, the database hierarchy is organized logically by category (group). All pyramids, obelisks, and trees are separated into second-level groups under the Egypt group node. The image generator must check each group node and each object node under the group nodes to find the objects that are in view.

For example, the image generator must traverse to the Pyramid group node and to each pyramid object node to isolate and display the pyramid in view. The image generator must find the obelisks and trees to display in the same manner. Again, cull performance slows down because the image generator must check every object.



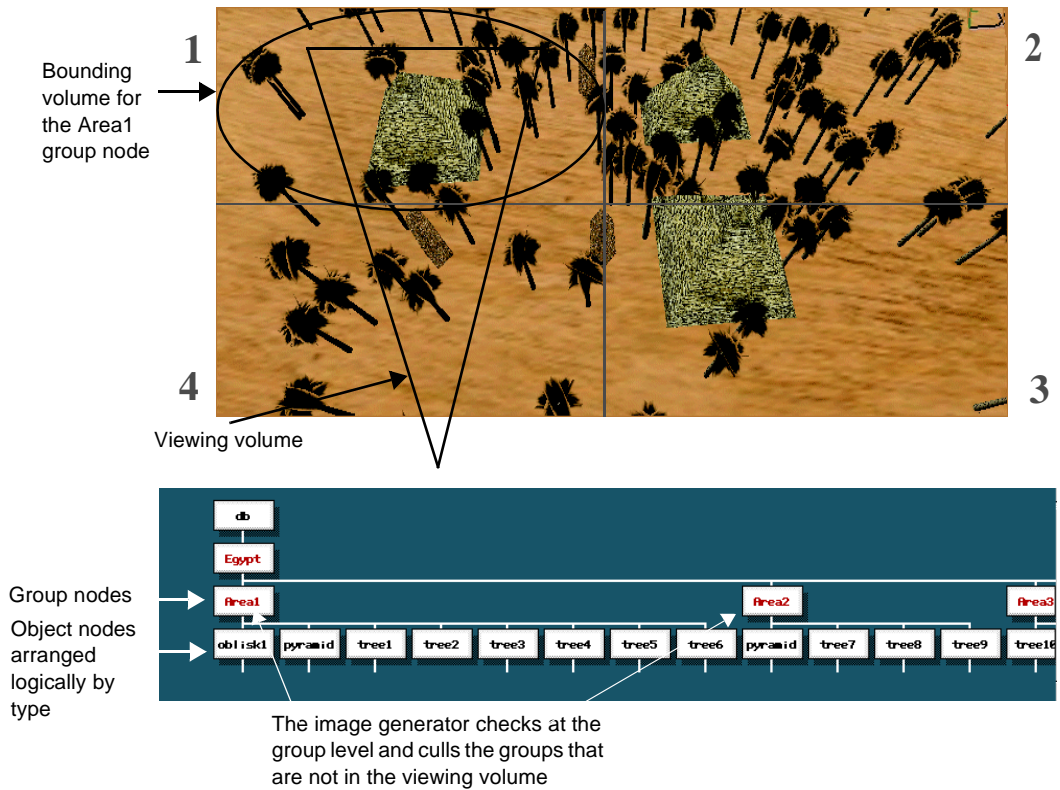
The image generator checks each object node to see if it is in the viewing volume

A logical structure might be practical for organizing the pieces of a model so that you can edit the nodes easily. For a large-scale database, however, this advantage is outweighed by the performance degradation that occurs when the runtime system must cull or draw every node.

Spatial Structure

With a spatial structure, the database hierarchy is organized into second-level groups, as in the logical structure, but all geometry is grouped by area in the database instead of by category. Most runtime systems perform faster when nodes are organized in groupings by their location in the database scene.

If the database were divided into quadrants as shown below, area 1 would contain a pyramid, obelisk, and trees; area 2 would contain a pyramid and trees; and so on.



This structure is the most efficient for large-scale databases because the image generator can quickly cull the areas that are not in the viewing volume by checking only the Area group nodes. If an Area group node is not in view, the image generator does not evaluate any of the hierarchy under this node. If an Area group node is in view, the image generator proceeds to its object node level to find and display the object nodes in view.

If geometry overlaps quadrant boundaries, the image generator must draw multiple quadrants to complete the drawing. You can either divide the geometry in half, with one node per quadrant, or physically move the geometry into one quadrant in the database. For simple objects like the trees or pyramids, it is probably easiest to move them.

Note: As you structure your database hierarchy, group all face nodes of the same color and texture together within object nodes to prevent state changes during runtime. State changes occur when a polygon with one set of attributes (states) switches to another polygon with different states in the database hierarchy. See “State Changes” on page 3-7 for more information.

Draw Order

In a 3D application, some models appear closer than others. If a closer object is between your eyepoint and a distant object, it should obscure the view of the distant object as it does in the real world. To achieve this effect, the closer model is drawn after the distant object is drawn. The order that models are drawn is called the *draw order*.

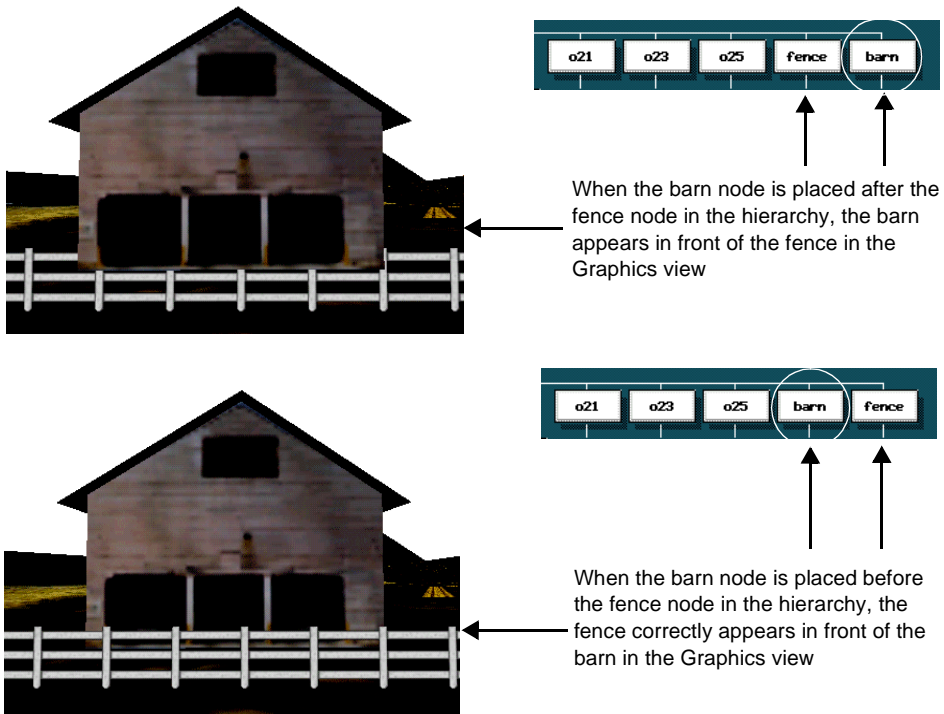
If you use a draw order correctly, the draw order can enhance the realism of a scene. If you use a draw order incorrectly, the draw order can detract from realism as well as slow the rendering in the runtime system. You design your database according to the type of draw order, *fixed list*, *binary separating planes (BSP)*, or *z-buffer*, that your runtime system uses. Most runtime systems use either a z-buffer or BSP draw order, but check before you design your database.

Fixed List

The default drawing order in Creator is *fixed list*, where each node is processed as the runtime system traverses the database hierarchy. Fixed list relies on the database hierarchy and *traversal order* to decide which geometry to draw first. The traversal order is the order in which the program processes the hierarchy. Creator reads the hierarchy from top to bottom and then from left to right.

Fixed list is the fastest drawing order, but might not always render a scene correctly. A node's geometry is always drawn in front of geometry in the preceding node. When modeling faces in front of other faces, you might need to transpose nodes in the hierarchy if the faces are drawn in the wrong order.

The following illustrations show that nodes needed to be transposed in the hierarchy to display a fence correctly in front of a barn using the fixed list drawing order.



Transposing nodes in the hierarchy would not help in the runtime if the eyepoint revolves around objects, however. Objects might still hide other objects. Binary separating plane (BSP) and z-buffer drawing order methods resolve this problem by processing according to the eyepoint. These drawing order methods are explained in

the next sections. Fixed list drawing might be appropriate if your runtime system does not have BSP or z-buffering capabilities, or if objects in your database are unaffected by a rotating eyepoint.

Binary Separating Planes

If you have a Binary Separating Plane (BSP) runtime system, you can add *BSPs* to a model so that the runtime system can quickly decide which geometry to draw before other geometry. BSPs are invisible planes that you can insert between separated geometry. Nodes are drawn according to their position relative to the BSPs. Nodes on the same side of a BSP are drawn in front of nodes on the opposite side of the BSP. After Creator finds the BSP, the geometry is drawn in a fixed-list draw order. Nodes that are not separated by planes are also drawn using the fixed-list draw order. Creator has a powerful BSP feature for automatically inserting BSPs in the database and for detecting problems when a database cannot be separated. You can also insert BSPs manually to customize the separation.



BSPs divide selected geometry and have a checkerboard pattern on their front side in the Graphics view. Geometry facing the front side of a BSP is drawn before geometry facing the back side.

Automatically Separating a Database

Creator automatically adds BSPs between *separable* sections in your database when you choose **BSP/Separate Selected**. A separable section can be divided. More precisely, a database is separable if a plane can be inserted between each pair of nodes without intersecting any geometry, and if none of the *convex hulls* around the items being separated are interpenetrating. A convex hull is the smallest convex region that completely encloses all the vertices of each item selected for separation. As you design your database, you must properly construct the models to be separable for a BSP runtime system. These are some design considerations:

- Plan the major divisions of each model before you draw it. For example, if you want major divisions in each direction, you must construct the model with many divisible sections. This requires more planning than if you want only one BSP down the middle of a model.
- Create symmetrical models, which can be divided in the middle. To build a symmetrical model, build one side, and then mirror the geometry across the tracking plane to build the other side using the Mirror tool in the **Modify Geometry** toolbox. Mirroring geometry not only saves drawing and correction time, but also ensures that Creator can insert a BSP down the middle of the model.
- Create tubular structures (such as airplane fuselages) with an even number of sides. Tubular structures with an even number of sides can appear to be asymmetrical when they are rotated. A major division with a BSP can be created in a model with an even number of sides.

When you choose **BSP/Separate Selected**, Creator separates the selected items, rebuilds the database hierarchy, and attaches each newly separated subtree to the *lowest common ancestor*. The *lowest common ancestor* is the first node, as you travel back up through the database hierarchy, that is an ancestor of each selected node. BSPs are added between the separated items.

Before you add the BSPs, you can choose **BSP/Check Separability** to create a temporary separation of the database, and preview the effect of **Separate Selected**. **Check Separability** creates temporary planes in the Graphics view of the database, showing the areas where the database will separate or fail to separate. No BSPs or BSP nodes are inserted. This lets you troubleshoot and correct portions of the database before inserting BSPs. For example, to correct interpenetrating hulls, you can move objects so that they do not intersect, or remove the part of geometry in one object that interpenetrates the other object. See the Creator online help for other techniques for correcting inseparable databases.

Other functions that you can use to automatically separate the database are **Separate Children**, which separates the children of each selected node without discarding nodes, and **Recursive Separation**, which traverses the selected portion of the hierarchy and separates parent nodes and their children while preserving intermediate levels of the hierarchy in the process.

Manually Separating a Database

You can add BSPs manually with the **Create BSP** tool and by choosing **BSP/Create BSP Plane**. The **Create BSP** tool inserts a BSP node under the current parent. The groups or objects that you want to separate become children of the BSP node and are separated with a BSP when you choose **BSP/Create BSP Plane**. If you use the **Slice** tool to separate geometry in the Graphics view, the sliced geometry are placed into separate nodes. You can then select the nodes and choose **BSP/Create BSP Plane** to insert a BSP between the geometry.

See the Creator online help for detailed instructions on automatically or manually creating BSPs.

BSP Example

This example of an L-shaped block and ball shows geometry that is not always drawn in the correct order as the eyepoint moves. BSPs were added to correct the problem. The first illustration shows the geometry without BSPs. Either the ball or a wall incorrectly draws before the other at certain angles.

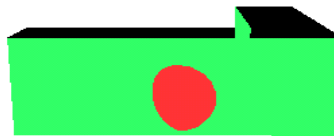
L-Shape and Ball Without BSPs



The L-shaped block and ball in the middle is not always drawn correctly as the eyepoint changes with only a fixed-list drawing method



At this angle, the wall is correctly drawn before the ball

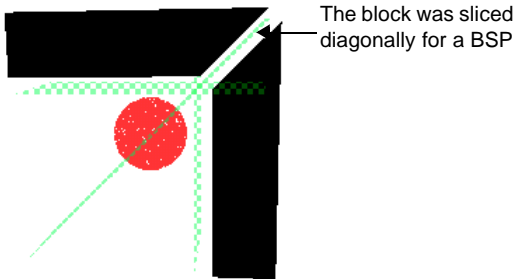


At these two angles, either the ball or a wall is incorrectly drawn before the other

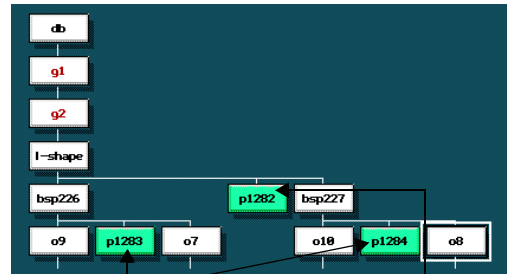


The second illustration shows the geometry with BSPs. Because BSPs cannot intersect geometry, the block was first sliced diagonally using the **Slice** tool and automatically placed into separate nodes as two separate walls. The ball is separated from both walls in horizontal and vertical directions with two additional BSPs. BSP nodes are placed between the divided nodes. As the eyepoint rotates, the ball and block draw correctly in all directions.

L-Shape and Ball With BSPs



Each wall is divided from the other with a BSP placed between them. The ball is also divided with the BSP.



BSP nodes are placed between the ball and both horizontal and vertical walls

A BSP node is placed between each wall (for the diagonal BSP)



After the BSPs are added, the ball and block are drawn correctly in all eyepoint directions

Z-Buffer

Models that are rendered using the *z-buffer* draw order are the easiest and fastest to create because you do not need to transpose database nodes or add separating planes to determine correct draw order. Instead, the z-buffer compares *pixel depths* of images to determine correct draw order. The pixel depth is the distance from the eyepoint to a point on the surface. The lower the pixel depth, the closer the object is to the viewer. Most graphics cards have z-buffer capability, where z stands for the z direction of depth.

The z-buffer is a block of memory equal to the display resolution, such as 640 x 480, multiplied by the number of bits of depth resolution of an image's pixels, such as 16 bits. The hardware rendering engine calculates a z value for each pixel in an image and stores the z value in the z-buffer memory location that corresponds to the x, y address of the pixel in the display buffer. When a new pixel's z-value is less than the pixel already stored, the new pixel's z-value overwrites the older pixel's z-value. The new pixel is written to the display buffer and displayed on the computer screen.

More memory needs to be allocated to render a scene with the z-buffer, however, and the z-buffer drawing method is inefficient when processing geometry with a high depth complexity or with a lot of polygons stacked behind each other.

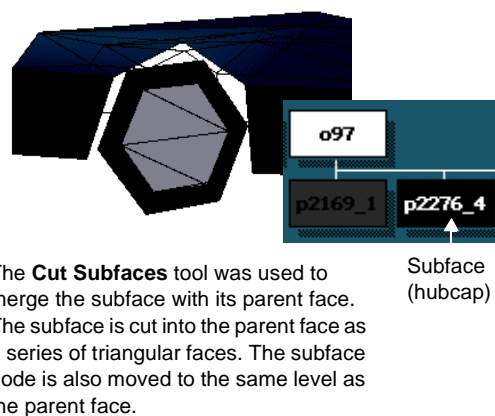
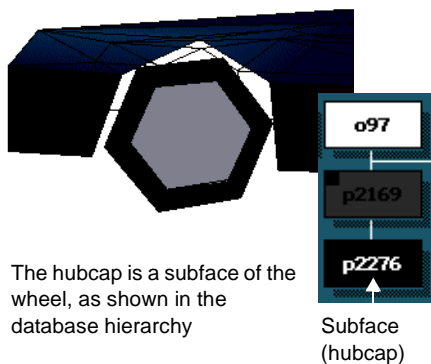
In Creator, you can enable z-buffer emulation by pressing the Z key. For the realtime application, the runtime system must have z-buffering hardware.

Problems with Coplanar Faces

A problem with the z-buffer drawing method is that it does not know how to resolve coplanar faces, where a face lies directly on top of another face. The condition called "z-fighting" occurs during runtime, when faces appear to flicker because both faces are at the same distance from the eyepoint and the z-buffer does not know which face to draw on top.

To resolve this problem, you can separate the faces into different nodes and move one node beneath the other node in the database hierarchy as a subface. The z-buffer draws all subfaces on top of parent faces.

If the pixel fill rate is too high, you can choose **Edit/Cut Subfaces** to merge subfaces into a selected parent face. The subface is cut into the parent face as a series of triangular faces, as shown in the following example. Subfaces and an additional level in the database hierarchy are removed, which improves performance on a z-buffer system. The polygon count increases, however, with the additional faces.



Using Bounding Volumes

A *bounding volume* is an invisible wireframe shape, such as a box or sphere, that surrounds a model. Bounding volumes that you define in Creator are used by some runtime systems to estimate the shapes of models for collision detection, culling data, or both.

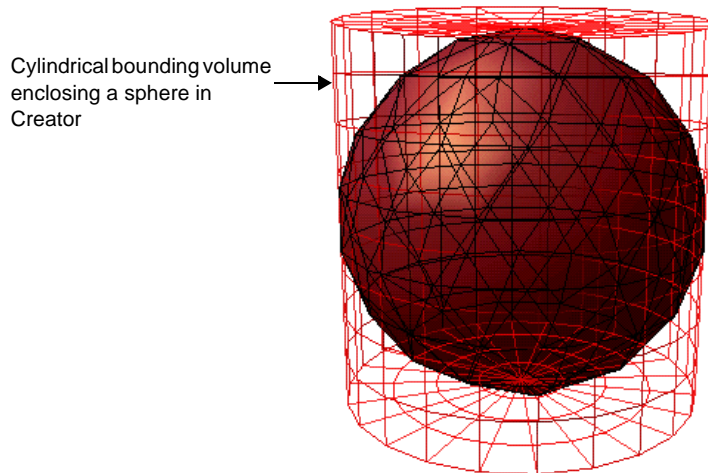
Collision detection occurs when bounding volumes on different models intersect. The runtime system responds as if the models collided. You can then program after-effects for your simulation, like sounds and explosions. Without bounding volumes, every polygon would need to be tested to see if it intersected with every other polygon in the simulation, and realtime speed would never be achieved. It is more efficient for the runtime system to test only the bounding volumes for collision.

Bounding volumes are also used for culling data in the runtime system. Group nodes are checked to see if their bounding volumes intersect the viewing volume. The viewing volume is the portion of the database that is currently visible. The runtime system traverses the group nodes only if their bounding volumes intersect the viewing volume. Nodes outside of the viewing volume are culled and not displayed.

Bounding volumes are attributes of group nodes and encompass all geometry below group nodes in the database hierarchy. You can set a bounding volume for a Group node, and choose the shape of the bounding volume that approximates the shape of the object, such as a box, sphere, or cylinder, in the *Group Attributes* window. You can also decrease the size of the bounding volume for tight collision with other objects or enlarge it for other effects. Bounding volumes become visible in the Graphics view when you choose the **Draw Bounding Volumes** option in the **View** menu.

Some runtime systems generate their own bounding volumes when you load your database. You can define bounding volumes in Creator, however, to be a different shape than the geometry. For example, instead of enclosing only the two blades of a helicopter's rotor in a bounding volume of a similar shape, you can define a cylindrical shape to represent the full movement of the rotor disk for the runtime system.

Bounding volumes are useful for estimating sizes of models, but too many bounding volumes in the database consume processing time and slow the speed of the runtime system. You should define bounding volumes only for objects that logically need them.



Reducing Polygons

The graphics hardware in an image generator imposes a limit for the number of polygons that can be translated, scaled, and rotated at a given frame rate. This limit is also referred to as the *polygon budget*. If the number of polygons exceeds the system capacity, frame processing is not completed in time to transfer into screen memory, and the illusion of smooth movement is destroyed.

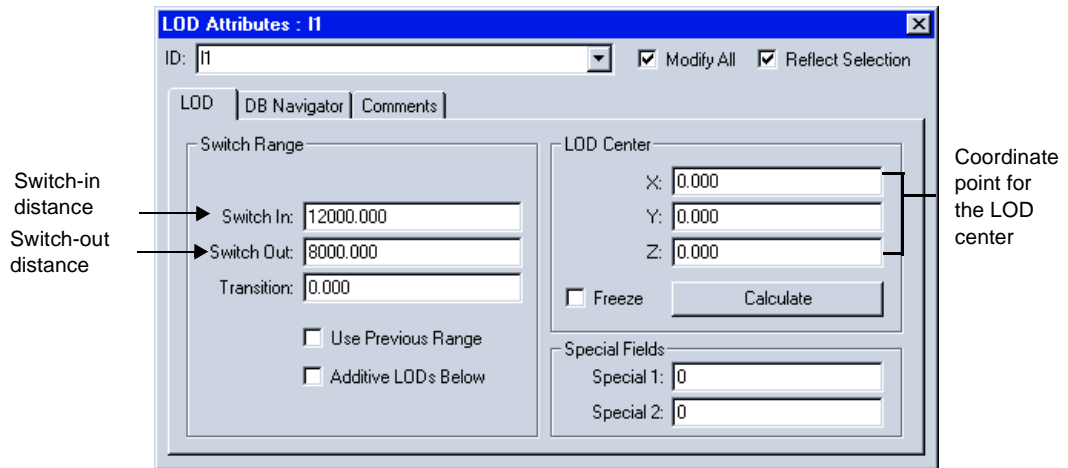
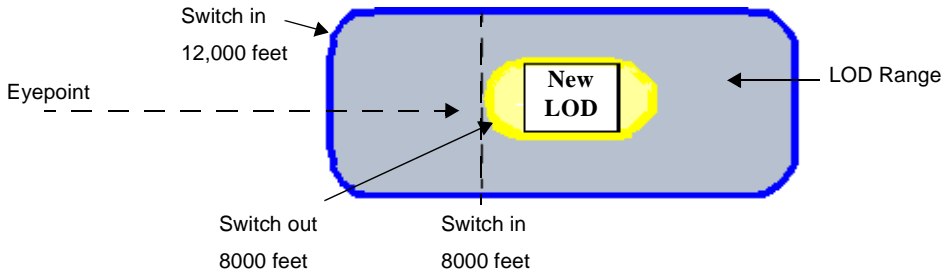
To reduce the number of polygons in your database, you can:

- Use levels of detail (LODs)
- Replace polygons with texture
- Remove unnecessary polygons
- Remove back faces of polygons when the back faces will not be visible in the runtime system

Using Levels of Detail

You can use *levels of detail (LODs)* to stay within the polygon budget and increase viewing performance. LODs are versions of the same model with different numbers of polygons. As the eyepoint moves closer to the model, more detailed versions are substituted. The version with the highest number of polygons, called the highest LOD, is only displayed when the eyepoint moves closest to the model in the runtime. When the eyepoint moves farther from the model, not as much detail needs to be visible, so a lower LOD is switched in.

You can control when each LOD is visible by setting *switch-in* and *switch-out* distances for the range between the eyepoint to the geometric center of the LOD, as shown in the following illustration. When you create an LOD, you set the switch-in and switch-out distances, and specify a coordinate point for the LOD's center in the *LOD Attributes* window.



When the eyepoint moves toward the LOD, the range becomes less than or equal to the switch-in value, and the runtime system displays the LOD. As the eyepoint continues to move towards the LOD, the range decreases. When the range becomes less than the switch-out value, the runtime system turns off the LOD. A higher LOD usually switches in at this point.

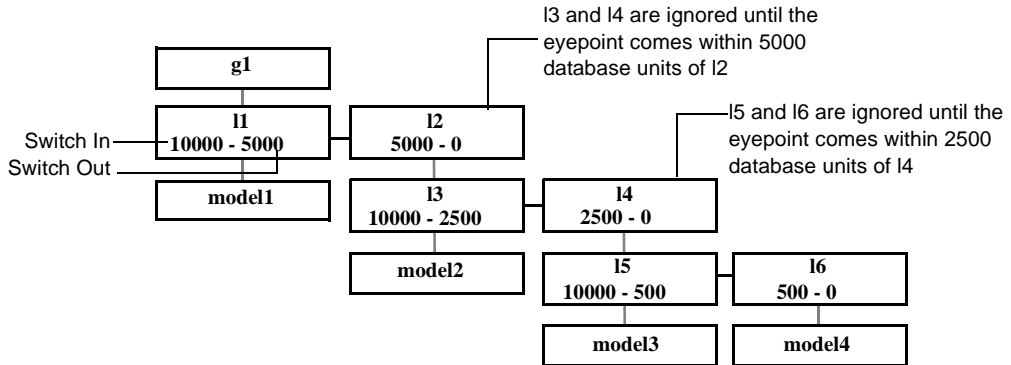
The following model of a car was created with three LODs. When the LOD with the least amount of detail is displayed, the other versions of the model are hidden. As the eyepoint moves in closer and the next switching distance is reached, another version of the car appears with more details.



A problem that can happen during runtime is visual discontinuity as one LOD switches to another. As the eyepoint moves in and out, the LODs do not change smoothly, which creates a “popping” effect. Setting a large switch-in distance (which directs an LOD to switch in when the viewer is far from the LOD) helps to prevent popping, but can decrease performance. You can *morph* LODs, which is the visual merging of one LOD into another, to prevent this problem. To morph LODs, you define certain vertices to display from the next lower LOD and use the original switch-in and switch-out distances. Performance is not affected as much as setting a large switch-in distance. See the Creator online help for directions for morphing between LODs.

LODs and Hierarchy Structure

You can structure the LODs in your hierarchy according to your type of database. For efficient display of a database with a lot of detailed models, LOD nodes are often nested in levels according to their switch-in distances. The following figure shows an example of a nested LOD structure.



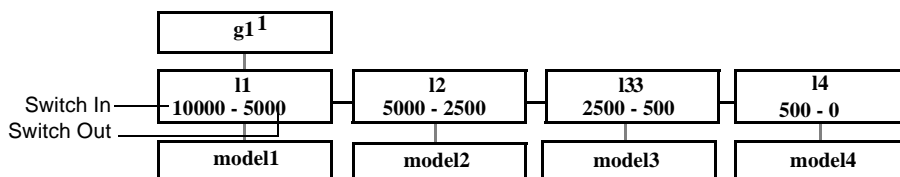
In the figure, **model1**, **model2**, **model3**, and **model4** are models attached to nested LODs that all represent the same item.

When the runtime system draws this database:

- The runtime system chooses to display **I1**, **I2**, or nothing, depending on the distance from the current eyepoint to the center of the LOD
- If **I2** is switched out, its descendants are ignored
- If **I2** switches in, the runtime system selects either **I3** or **I4** to display
- If **I4** switches in, the runtime system selects either **I5** or **I6** to display

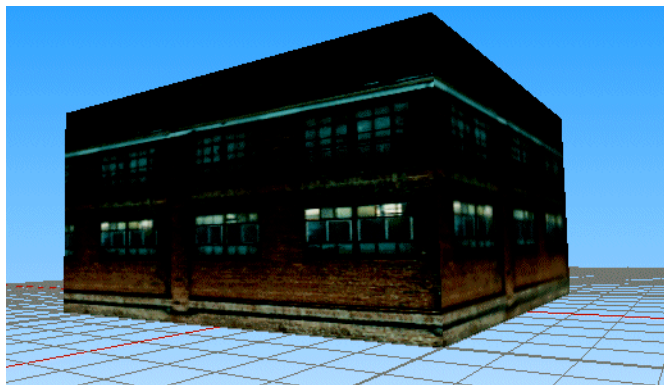
The LODs of the models are all descendants of the parent nodes. In this way, the runtime system only needs to switch in the parent node and its descendants when the eyepoint reaches the corresponding distance from the model. The runtime system culls LODs that are not attached to the LOD node selected for display.

You can arrange a flat hierarchy with LODs if your models do not have a lot of detail. Each LOD is selected for display only for its switch-in and switch-out attributes, as shown in the following figure. The runtime system, however, must test each node for display.



Replacing Polygons with Texture

To add detail to the scene without adding polygons, you can map textures on models. For example, in the following illustration, each face of a box has a mapped texture to represent a side of a building.



You can also apply textures with different resolutions on versions of the same model. These different versions are assigned in LODs that switch as you zoom in and out. The version seen at close range can have a high-resolution texture applied, and the version seen farther in the distance can have a low-resolution texture applied. A texture resolution for a scene visible at three meters or greater is approximately 0.005 meters per texel. For 100 meters or more, a lower-resolution texture is usually 0.164 meters per texel. Using LOD models with detailed textures instead of adding polygons to models can significantly reduce the number of polygons that the image generator needs to draw.

Removing Unnecessary Polygons

You can reduce the number of polygons in your database by manually deleting polygons that will not be seen in the runtime system. These polygons can be details inside of models, polygons behind other polygons, or the bottoms of models that are set on top of polygons, such as the bottom of a house on the ground, for example.

You can also use Virtue 3D, Inc.'s VSimpleify automatic polygon reduction tool to reduce polygons. With this tool, you can automatically reduce the number of polygons by a specified percentage while retaining properties such as textures, normals, and colors on the model. VSimpleify installs as a plug-in to Creator and usually appears in the LOD menu. If VSimpleify does not appear in the LOD menu, choose **Help/On Plugins...** to open the **MultiGen Creator: Help On Plugins** window. When you select the VSimpleify plug-in in the **Plugin Modules** list, its location appears on the bottom of this window.

Removing Back Faces of Polygons

Creator automatically culls the back faces of polygons and displays only the front faces. Culling each back face that you do not need helps to reduce the runtime system's drawing time. For example, to achieve a realistic effect when flying over a building, only the outside of the building needs to be drawn. The inside face of each wall, including the bottom face, do not need to show in the runtime.

If you want to display both sides of a face, you can choose the **Render Both Sides Visible** attribute in the *Face Attributes* window, which is retained by runtime systems that use the Performer loader. The back face has the same color, texture, and other characteristics that are applied to the front face.

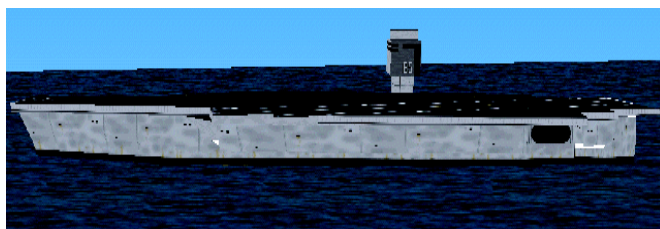
If you are planning to fly in the interior of the building, inside faces (back faces) usually need a different texture than the outside faces, so displaying the back faces would not be effective. In this case, you can select the front face of a polygon, duplicate it, and apply the polygon to the inside face to make two separate polygons that appear to be attached. Different textures could then be applied to the inside polygon and the outside polygon. For each of these two polygons, you want the back face removed, which is the default; otherwise, the two polygons become coplanar and z-fighting can occur on a z-buffer system. With z-fighting, the two polygons and their textures would flicker because Creator (and the runtime system) would not know which polygon to display first.

Moving Clipping Planes

You can adjust *clipping planes* to control the geometry that appears in the Graphics view. Clipping planes define the near (inner) and far (outer) limits of the viewing volume. The viewing volume is the portion of the database that is visible. Objects that lie outside of the clipping planes are “clipped” from the scene and not drawn. The computational requirements of drawing a large database or using z-buffering can slow the speed of the display in the Graphics view. Moving clipping planes can increase viewing performance.

The near clipping plane in front of the viewing volume does not need to coincide with the screen’s position, and both the near and far clipping planes can be repositioned at any depth using the clipping plane scale in the *Viewing Volume* dialog box (**View/Viewing Volume**). The clipping planes are not transferred into the runtime system, and are meant for either emulation or viewing purposes in Creator.

Clipping planes are always in the database, but you can adjust their positions to cull parts of objects that you do not expect to be rendered in the runtime version. For example, if a certain part of an aircraft carrier will not be visible at 500 nautical miles from the eyepoint in the runtime, you can preview your final scene in Creator by setting a far clipping plane to 500 to cull that part out of the viewable scene.



No parts of the aircraft carrier are culled using clipping planes



Parts of the aircraft carrier are culled using a far clipping plane

Clipping planes are also useful for culling certain geometry when you are trying to display a large database on a system that has slow display performance or on a z-buffer system that uses a lot of memory. See the online help for the **View/Viewing Volume** command and detailed procedures for changing clipping planes.

Index

Numerics

- 3D Graphics
 - compared with 2D 1-3
 - defined 1-3
 - tessellation 1-3
- 3-Point Put tool 4-12
- 4-Point Put tool
 - aligning a texture with a corner 4-13
 - using 4-13

A

- ADF (application definition file) 1-6
- Ambient intensity
 - bidirectional lights 4-48
 - unidirectional lights 4-48, 4-49
- Ambient lighting
 - changing intensity 4-36
 - defined 4-34
 - materials 4-41
- Angular tolerance
 - shading effect 4-39
 - vertex normals 4-39
- Animations
 - compared with realtime applications 1-2
 - defined 1-2
- Application definition file (.adf) 1-6
- Approximating spline
 - B spline 5-6
 - Bezier spline 5-6
- Attenuation
 - attenuation factor 4-37
 - defined 4-37
- Attenuation factor
 - constant 4-37
 - distance 4-37
 - Intensity@distance 4-37
 - linear 4-37
 - quadratic 4-37

B

- B spline 5-6
- Back faces 6-22
- Back-facing color 4-50
- Background images 5-8
- Balancing culling and drawing 3-4
- Bezier spline 5-6
- Bidirectional lights
 - ambient intensity 4-48

defined 4-48

- Billboards
 - creating 5-9
 - transformation matrix 5-9
 - using with instances 5-9
- Binary Separating Planes (BSP)
 - automatically adding 6-10
 - convex hulls 6-10
 - creating 6-9
 - database structure issue 2-7
 - example 6-12
 - manually adding 6-11
- Blending multitextures 4-21
- Bounding volumes
 - collision detection 6-15
 - cull process 3-3, 6-15
- Bright light
 - lighting effects 4-43
- BSP - see "Binary Separating Planes (BSP)"

C

- Calculate Shading process
 - Flat shading model 4-39
 - Gouraud shading model 4-39
 - lighting effects 4-38
 - Lit Gouraud shading model 4-39
 - Lit shading model 4-39
 - vertex normals 4-38
- Calligraphic mode 4-50
- Cardinal splines 5-5
- Clipping planes
 - culling objects 6-23
 - in the runtime system 3-5
 - z-buffer system 6-24
- Collision detection 6-15
- Construction curves
 - approximating spline 5-5
 - control points 5-4
 - interpolating spline 5-5
 - uses 5-4
- Construction edges 5-3
- Convex polygons 1-3
- Cookie Cutter tool 4-43
- Coplanar faces
 - and z-buffer system 6-14
 - avoiding 4-10
- Coplanar vertices
 - importance for modeling 4-9

- triangles 1-4
- Cull process
 - bounding volumes 3-3, 6-2, 6-15
 - organizing a database hierarchy for 6-2
 - realtime application process 3-3
 - viewing volume 3-3, 6-2
- Curves
 - approximating splines 5-4
 - Cardinal spline 5-5
 - control points 5-4
 - interpolating splines 5-4
- D**
- Database elements 2-2
- Database header node 2-3
- Database hierarchy
 - draw order 6-7
 - linear structure 6-3
 - logical structure 6-5
 - nesting LODs 6-20
 - organizing for cull process 6-2
 - purposes 2-1
 - Pyramid database example 6-3
 - spatial structure 6-6
- Database node organization
 - example 2-3
 - issues to consider 2-7
- Degrees of Freedom (DOF)
 - adding 4-52
 - checking movement 4-55
 - database component 2-4
 - database structure issue 2-7
 - defined 2-4
 - DOF limits 4-54
 - local coordinate system 4-52, 4-53
- Depth complexity
 - defined 3-6
 - increase and effect on draw performance 3-6
 - methods for reducing 3-6
 - pixel fill rate 3-6
- Developing a realtime application 1-5
- Diffuse lighting
 - changing intensity 4-36
 - defined 4-35
 - increasing with Cookie Cutter tool 4-44
 - materials 4-41
- DOF - see "Degrees of Freedom (DOF)"
- Draw order
 - BSPs 6-9
 - fixed list 6-8
 - z-buffer 6-14
- E**
- Emissive lighting 4-41
- Environment Map Texture tool 4-17
- External references
 - applying a transformation edit 5-13
 - creating 5-10
 - defined 5-10
- F**
- Face node
 - defined 2-3
 - location in the database hierarchy 2-4
- Fixed list drawing order 6-8
- Flat shading
 - Calculate Shading process 4-39
 - defined 4-39
 - example 4-41
- Front-facing color 4-50
- Frustum
 - clipping planes 3-5
 - defining 3-5
 - in the runtime system 3-5
- G**
- Geometry, defined 2-2
- Global database system 5-1
- Gouraud shading
 - Calculate Shading process 4-39
 - removing light sources after shading 4-39
- Graphics card
 - number of textures supported 4-26
- Group node 2-3
- H**
- Hierarchy
 - defined 2-2
 - issues to consider for structuring 2-7
 - node organization 2-4
 - pyramid example 2-4
- I**
- Image generators 1-6
- Infinite light source 4-30
- Instances
 - advantages/disadvantages 5-11

- applying a transformation edit 5-13
 - creating 5-11
 - defined 5-11
 - using with billboards 5-9
- Intensity, texture pattern 4-11
- Intensity@distance 4-37
- Intensity-alpha, texture pattern 4-11
- Internal data format, reducing 4-24
- L**
- Latency period 1-2
- Levels of Detail (LOD)
 - applying texture to 4-20
 - database component 2-4, 2-7
 - defining light points 4-52
 - morphing 6-19
 - nested LODs 6-20
 - reducing number of polygons 6-17
 - switch-in distance 6-18
 - switch-out distance 6-18
- Light intensity, changing 4-36
- Light point lobes 4-51
- Light points
 - bidirectional lights 4-48
 - calligraphic mode 4-50
 - defined 4-46
 - front-facing color, back-facing color 4-50
 - LODs 4-52
 - node 4-46
 - omnidirectional lights 4-47
 - raster mode 4-50
 - types 4-47
 - unidirectional lights 4-48
 - viewing direction 4-47
- Light sources 2-4
- Lighting
 - ambience 4-34
 - applying 4-29
 - Calculate Shading process 4-38
 - diffuse 4-35
 - infinite light source 4-30
 - local light source 4-31
 - modeling light source 4-32
 - omnidirectional lights 4-47
 - shading 4-27
 - specularity 4-36
 - spot light source 4-31
 - vertex normals 4-38
 - with materials 4-41
- Linear curve 4-38
- Linear database structure 6-3
- Lit Gouraud shading model 4-39
- Lit shading model 4-39
- Local coordinate system
 - defining with a transformation 5-2
 - for DOFs 4-52
 - moving an object 5-3
 - when to use 5-1
- Local light source
 - attenuation 4-37
 - defined 4-31
- LOD - see "Levels of Detail (LOD)"
- Logical database structure 6-5
- LynX utility 1-6
- M**
- Magnification tool 4-12
- Map Texture tools 4-11
- Materials
 - and lighting 4-41
 - blending with textures 4-42
- Minification tool 4-12
- Mode 2-6
- Modeling light source 4-32
- Modeling with polygons 4-9
- Morphing LODs 6-19
- Multitextures
 - applying 4-21
 - blending 4-21
 - reducing texture memory 4-21
- N**
- Nested LODs - see "Levels of Detail (LOD)"
- Nodes
 - organization in database hierarchy 2-4
 - selecting modeling mode to create 2-5
- Nonplanar faces
 - correcting 4-9
 - OpenGL requirements 1-4
- O**
- Object node
 - defined 2-3
 - location in the database hierarchy 2-4
- Omnidirectional lights 4-47
- OpenFlight 1-1, 1-5

OpenGL

- polygon requirements 1-3
- support for Creator 1-5

Orthographic projection 3-5

P

Performer loader 1-6, 3-2, 6-22

Perspective projection 3-5

Pixel fill rate

- defined 3-6
- z-buffer system 6-15

Pixels

- overview 4-11
- texel mapping 4-11

Planar faces

- creating 4-7

Plug-ins, locating 6-22

Polygon budget 6-17

Polygons

- applying texture to reduce number 6-21
- invalid 1-3
- reducing with LODs 6-17
- reducing with the VSimpleify plug-in tool 6-22
- removing back faces 6-22
- simple, convex 1-3
- tips for creating 4-9

Project tool

- correcting nonplanar faces 4-9
- with tracking plane 4-5

Q

Quadratic curve 4-38

R

Radial Project Texture tool 4-16

Raster mode 4-50

Ray-tracing 4-18

Reality Engines 1-6

Realtime application process 3-3

Realtime applications

- and Creator 1-1
- basic development process 1-5
- compared with animations 1-2
- defined 1-2
- types 1-2

Rendering process 3-2

Repetition Factor

- Radial Project Texture tool 4-16

Spherical Project Texture tool 4-15

Surface Project Texture tool 4-15

RGB texture pattern 4-11

RGB-alpha texture pattern 4-11

Runtime performance 5-13

Runtime system

- culling and drawing 3-4
- defined 1-6

S

Scale tool 4-5

Shading

- angular tolerance 4-39
- creating flat 4-39
- creating smooth 4-39
- shading models 4-39

Shadows

- lighting effects 4-42

Simple polygons 1-3

Sky colors 4-29

Slicing geometry 4-4

Smooth shading

- creating 4-39
- example 4-40

Sound

- files 2-4
- loading 4-56
- node 4-57
- tasks to add 4-55

Spatial database structure 6-6

Specular lighting

- changing intensity 4-36
- defined 4-36
- increasing with Cookie Cutter tool 4-44
- materials 4-41

Spherical Project Texture tool 4-15

Spot light source

- attenuation 4-37
- defined 4-31
- defining the cone of light 4-31

Spotlights

- lighting effects 4-44

State changes

- effect on draw process 3-7
- moving nodes in hierarchy 3-7

Subfaces

- for z-buffer systems 6-14
- removing 3-6

- with coplanar faces 6-14
- Subtextures
 - applying to reduce texture memory 4-25
 - creating a collage 4-26
- Surface Project Texture tool 4-15
- Switch-in distance 6-18
- Switch-out distance 6-18
- T**
- T vertices
 - avoiding 4-10
 - correcting 4-10
- Tessellation 1-3
- Texels
 - mapping to pixels 4-11
 - overview 4-11
- Texture Environment algorithms 4-22
- Texture Mapping palette 4-18
- Texture memory
 - applying subtextures to reduce 4-25
 - calculating for a texture 4-24
 - reducing internal data formats 4-24
 - using multitextures to reduce 4-21
- Texture patterns
 - intensity 4-11
 - intensity-alpha 4-11
 - RGB 4-11
 - RGB-alpha 4-11
- Textures
 - applying on two sides of a face 6-22
 - applying to multiple LODs 4-20
 - applying to reduce number of polygons 6-21
 - blend algorithm 4-22
 - blending with materials 4-42
 - decal algorithm 4-22
 - graphics card support 4-26
 - mapping with tracking plane 4-7
 - modulate algorithm 4-22
 - replace algorithm 4-23
- Tracking plane
 - mapping textures 4-7
 - modifying geometry 4-5
 - placing items onto geometry 4-3
 - positioning on axes 4-2
 - tasks for using 4-1
 - using Rotate Grid controls 4-3
 - using to slice geometry 4-4
- Transformation edit

- applying 5-12
- using a transformation matrix 5-12
- Transformation matrix
 - defined 5-12
 - using with billboards 5-9
- Triangles
 - coplanar vertices 1-4
- Triangulate tool 4-9
- U**
- Unidirectional lights 4-48, 4-49
- V**
- Vega 1-1, 1-5, 1-6
- Vertex node 2-3
- Vertex normals
 - angular tolerance 4-39
 - Calculate shading process 4-38
- Viewing volume
 - clipping planes 3-5
 - cull process 3-3
 - defined 3-5
 - in the runtime system 3-5
 - orthographic projection 3-5
 - perspective projection 3-5
- VSimplify plug-in tool
 - location in Creator 6-22
 - using to reduce polygons 6-22
- Z**
- Z-buffer
 - clipping planes 6-24
 - defined 6-14
 - problems with coplanar faces 6-14
 - z-fighting 6-14
- Z-fighting 6-22

