CHAPTER 1

Basic Game Art Concepts

Some people don't want to take games seriously. This isn't a lament against parents lashing out over gore or people who think that games are a waste of time. It is instead a statement about a popular assumption that making video games doesn't involve a lot of work. This assumption is a problem in many game schools, whether they focus on the art, programming, or overall design of games.

In many ways, the art produced for a video game is not "art" in the traditional sense of "fine art"—content created solely for enjoyment by the senses or for aesthetic value. The art created for video games is, rather, part of the process of design or "applied art." But there are two important reasons why video game art is different from either animation or fine art. First, game art is interactive for a player. Second, game art is rendered in real time.

Many quests begin not with an instant foray into death and danger, but instead with an initial gathering of knowledge. Indeed, Sun Tzu's *The Art of War* does not begin by discussing battle right away, but with a chapter on planning. This is how we will approach your own quest to be a game character artist—and perhaps your leap from game consumer to creator.

This chapter covers the following topics:

- **Game design workflows**
- **Creating game assets**
- **Understanding and optimizing 3D game art**
- **Working with game engines**
- **Scripting happens**

Game Design Workflows

As a young field, game design has no prescribed "workflow" or "design method." This fact means that game studios can establish their own ways of working—within certain parameters, of course. Unless you are the rare person who can program like John Carmack and draw like da Vinci, you will most likely be working with a team. In many ways, this is one of the current strengths of the gaming industry: Game designers come from all fields and all walks of life. Consequently, team members have a broad range of influences to pull from that helps them create the best game possible.

Of course, games have to be fun. Game designers test this aspect with *playtesting*: inviting people to play early versions of a video game. Typically, games may see outside playtesters coming in at a stage of near completion known as the "beta" stage. At this point, many of the mechanics are already set in stone and the game has its artwork added to the engine. Because of this, some designers urge playtesting earlier in the process.

Other "set in stone" parts of the game design process involve the business end of things. This aspect is most directly embodied in the relationship between developers and publishers. The development team, which puts the game together, can consist of programmers, artists, writers, testers, producers, and composers—anyone involved in the legwork of making the game. Publishers oversee financial affairs, legal issues, public relations, and marketing of the game, turning it into a viable product.

Projects are often structured in milestone-based schedules, where the publisher pays the developer in phases. These phases generally include:

Concept Concept planning, budget, and contract negotiation

Preproduction Prototypes, design documentation, sketches, and basic design

Production

- **"Alpha" Development** Assets, levels, and early code
- **Beta/Quality Assurance** Playtesting with outside testers, utilizing beta code, and trying to reach launch or "gold" status by eliminating bugs
- **Gold** The point where the game is ready for mass production and retail and review copies are sent out to press

Beyond these phases, the developer has freedom to follow their own design methods as long as their obligations to the publishers are met. This has led to several distinctive design methods employed by studios; we'll look at some useful ones next. Try them in your projects and mix and match as you find appropriate.

Phase-Based Design

Phase-based design is a literal interpretation of the milestone-based schedule employed by many game publishers. It is also one of the most commonly used methodologies in the

industry. In this process, departments operate separately but on a linear schedule that works toward final release of a game. The game goes from early concept stages to the production, or "alpha," level. Playtesters are then brought in to play and evaluate the game in the "beta" stage of development. Finally, the game is polished and sent for release, the "gold" stage.

Although many great games have been developed using phase-based design, other games have been deemed failures, often due to the separation of departments trying to create a cohesive product and the lack of early playtesting. While an industry standard, designing in this way also runs the risk of the design team suffering from poor communication and developing a game to near completion before someone realizes that it is no fun to play! Individual studios have, over time, developed their own methods for resolving these issues. All of the following methodologies follow the "alpha, beta, gold" standard but differ from phase-based design by attempting to address its intrinsic weaknesses.

The Cabal

Valve Corporation employees created this method while working on the original *Half Life.* The Cabal is a collaborative design method where several representatives of various design departments, such as programming, writing, animation, and level design, work closely on one part of a game. The crux of this method is teamwork among departments that may otherwise work separately. In this way, game elements fit together in a more cohesive manner and mechanics work together with art to create a livelier world.

This process is further discussed in the Gamasutra essay, "The Cabal: Valve's Design Process for Creating *Half Life*" (`http://www.gamasutra.com/view/feature/3408/the_cabal_valves_design_process.php`).

Playcentric Iterative Design

In her book *Game Design Workshop* (Morgan Kaufmann, 2008), game designer Tracy Fullerton describes a design process that initiates playtesting as early as possible. Fullerton argues that designers, upon conception of a game idea, should decide what the basic gameplay mechanics will be and create paper prototypes right away. The reasoning of this method is that if core gameplay of a game is fun, then it is worth developing. Further prototypes are developed as features are added to the game. Fullerton argues that a game should pass through several phases of paper prototyping before a design document is even written. In this way, the document can become a more effective guide for the further developing of the game.

The structure of this method is based on iterative design methods of software development, where software is prototyped and then tested. If any bugs or difficulties of use are found, they are documented and another phase of rebuilding and testing is done.

Naturally, as fewer things are fixed, or as fewer large-scale things are fixed, the game progresses to its final gold state.

This selection of design methods offers a sample of how studios customize their design process to create better games. While this book focuses on art asset creation, it is helpful to understand how individual *assets*, or pieces of art that go into the game, fit into the workflow of a whole game. If you understand how your art will be used, the transition from Blender to a game engine will be easier.

Now that you have an overview of the game design process, it is time to look at how art assets are planned and created.

Creating Game Assets

Creating characters is an intricate part of the concept process. Good character art is the result of not only interesting artwork, but often good writing. In line with Fullerton's process for good game design, much of the work for character development is done on paper, where the character's personality is developed. When planning characters, try to come up with ideas for the character's family, background, tastes, political and religious beliefs, and possibly even more personal/risqué parts of what makes them who they are. Although the written part of character creation seems unrelated to the art portion, the character's look can depend greatly on personal preferences established in your text. A character's animation may eventually be impacted by this information, as the way he or she moves and reacts will be determined by personality traits.

Creating Concept Art

When a character is well defined in writing, their look is defined with a series of perspective or action drawings. Main characters obviously require a lot of design, not only to make them visually engaging but also to make them marketable. Other characters can require varying degrees of elaboration.

The role of concept art in game asset production cannot be downplayed: Concept art gives designers an educated idea of what to produce. It also establishes the look of a game well before asset production has started. Even for basic characters, an understanding of the game's look in 2D can be incredibly helpful once 3D work begins. It can be difficult to work from 3D right away when it comes to character art.

Digital Painting

A key method of concept art creation is digital painting. Digital painting is the act of using the paint tools of a photo-manipulation program to create conceptual imagery. Examples of this type of software include Adobe Photoshop, GIMP, Corel Painter, and

ArtRage Studio Pro. Digital painting utilizes features of art programs such as layers, filters, and burn and dodge tools to make the process of painting much faster than with traditional real-world media. These paintings can illustrate a great deal of potential game elements and provide a clear vision of what the game will eventually look like. Things that benefit greatly from being designed as digital paintings include

- Characters
- First-person views of potential game levels/locations
- Weapons
- Vehicles
- Environmental objects
- Storyboards
- Model sheets
- UV layouts

Figure 1.1
Digital painting for a level

As you can see, this list describes nearly everything in a game. Digital painting is an effective method for creating concept artwork if you have the time for a full development cycle. Figures 1.1, 1.2, 1.3, and 1.4 show several examples of digital paintings used to conceive game elements.

Figure 1.2
Vehicle design

Figure 1.3
Character design

Figure 1.4
Enemy design

Model Sheets

Model sheets (Figure 1.5) are an incredibly important aspect of character development that will save you a lot of work and frustration. Blender and other 3D programs have features that allow you to view their model in a wireframe or x-ray mode so that if you are modeling from model sheets you can essentially trace in 3D.

There are several ways to view the process of creating model sheets. Nearly everyone agrees on several things: that characters should have their feet shoulder-width apart, that there should be front and side views, and that arms should not be down against the character's torso. These guidelines help create models for usable characters. In 3D modeling you have to be able to see the polygons that you are working on; this is why you need to pose the legs and arms. These two guidelines also help in eventual animation, as a character with better-defined groin and armpit geometry will animate more naturally. The front and

side views are there to show the character from multiple angles and eliminate guesswork. In fact, there is nothing that says you cannot create a back, top, or bottom view. Whatever will make your life easier is what you should have as a model sheet view.

Figure 1.5
Model sheet

Artists have differing opinions about how to position the character's arms and whether or not the model should appear relaxed. Many model sheets show the character in what is known as a "t-pose," with the arms positioned straight out from the sides. Others have arms that bend down slightly in a more relaxed position. This is often paired with relaxed hands and slightly bent knees. The reason for this is that when the character is animated, they will appear less stiff and robotic when they move. While this is great, keep in mind that the major advantage the t-pose gives you is the ability to model geometry at 90-degree angles. You don't need to take extra time to do odd rotations when working this way.

A good method is to work with a mixture of both methods for the arms. Create your model sheet and initial geometry with a t-pose and later rotate the arm geometry and hands to be in a slightly more relaxed position in 3D.

Figure 1.6 shows the finished model against the t-pose model sheet.

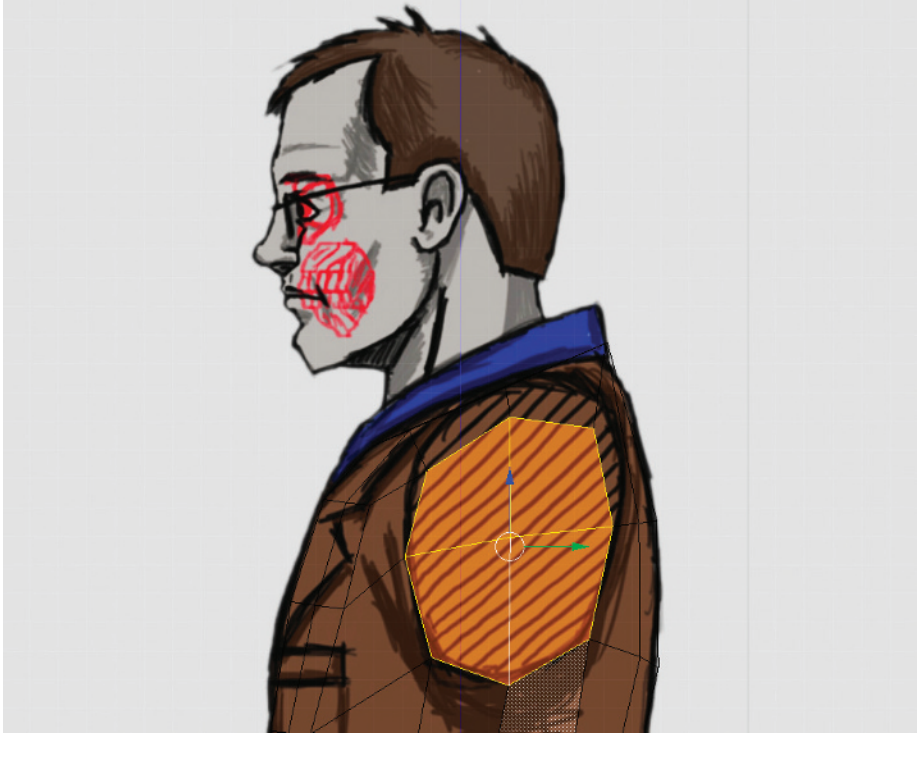
Another potentially useful precaution is removing the arm from the side view of the model sheet. Sometimes arms and hands can get in the way of details on the sides of the torso. When modeling the basic torso model, having a "chopped off" view of where the arm will be even helps create a nice template for the model's shoulder geometry (Figure 1.7).

Figure 1.6

Character t-pose sheet vs. the relaxed model

Figure 1.7

In this view, the highlighted portion shows the shoulder geometry being traced around the shoulder lines on the model sheet.



Understanding and Optimizing 3D Game Art

The logical progression from model sheets is to begin work on the 3D character model. When it comes to 3D art, game artists have it comparatively rough compared to other 3D artists. The next few sections of this chapter will discuss the differences between the two fields and what game artists can do to optimize their work.

3D Object Construction

You may be familiar with the xy coordinate plane, a system of two axes, one going horizontal (x-axis) and one going vertical (y-axis.) If you imagine that these axes each have some unit of measurement along them that can be indicated numerically, you can describe where certain points are within the space of the coordinate plane as an expression of where they are on x and y (Figure 1.8).

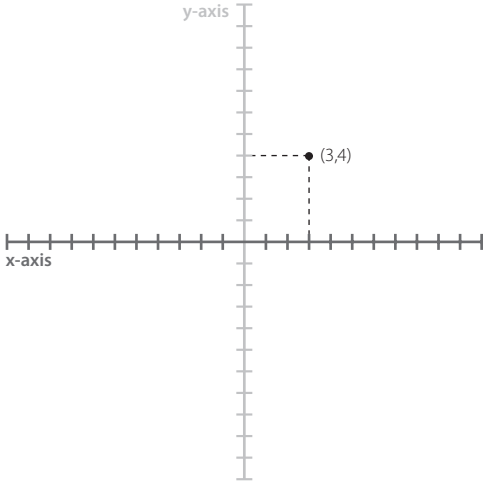


Figure 1.8
This point can be said to be at point (3,4) on the coordinate plane because it is at 3 units on the x-axis and 4 on the y-axis.

If you were to draw another point on the plane and connect them, you would create a line (Figure 1.9). When multiple lines are created on the plane and arranged into closed shapes, the resulting two-dimensional shapes are called "polygons." The simplest of these is a triangle, consisting of three lines (Figure 1.10).

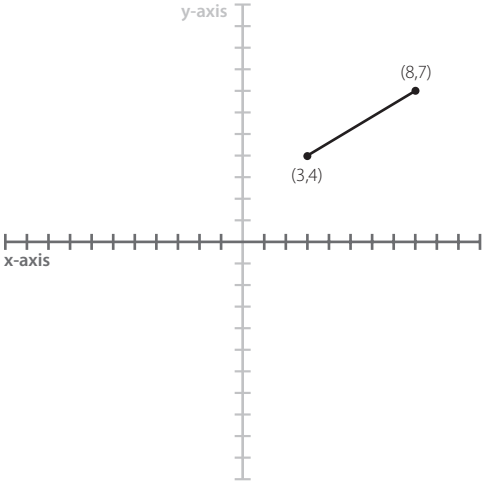


Figure 1.9
Line on the coordinate plane

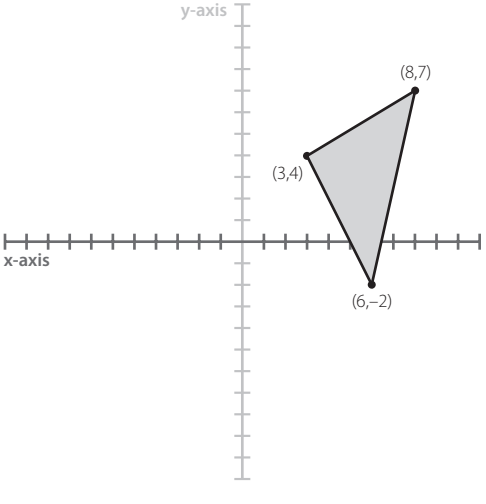


Figure 1.10
Triangle on the coordinate plane

Now let's add a third axis, the z-axis. If shapes on the xy coordinate plane can be said to be two-dimensional, objects on the xyz plane are three-dimensional. Points and lines have three-part coordinates consisting of their location in relation to the x-, y-, and z-axes (Figure 1.11). Also, like their 2D counterparts, polygons exist in 3D along these axes and can be combined to create 3D forms such as cones, cubes, spheres, and many others, as shown in Figure 1.12.

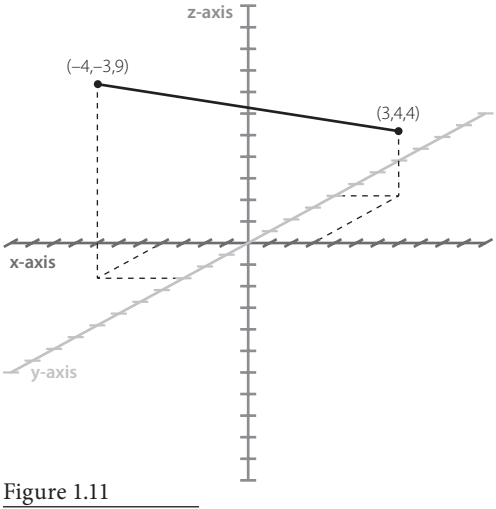


Figure 1.11
The xyz coordinate plane with a line

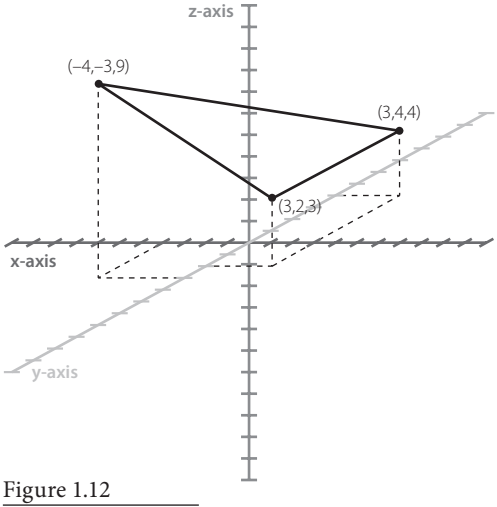


Figure 1.12
The xyz coordinate plane with a polygon

These types of objects and measurement systems are the basis of what we know as 3D computer graphics, also known as computer-generated imagery (CGI). When you work with objects in 3D modeling programs like Blender, you manipulate the points, lines, and polygons of an object to create complex forms.

In the computer graphics industry, the points of a model are called *vertices* (*vertex* in the singular). The lines between vertices are called *edges*. The three-dimensional forms that are created when at least three vertices are connected by edges are called *polygons*. If a polygon has a surface on it, this is called a *face*. Figure 1.13 illustrates these concepts.

When polygons or faces are arranged in such a way that they create a three-dimensional form, this is called a *mesh*. Some meshes of basic 3D forms are so widely used that they come premade in 3D programs. These are called *primitives*. The objects in Figure 1.14 are some of the basic mesh forms available in Blender. Many are primitive objects, though there are some more complex ones like Suzanne the Monkey.

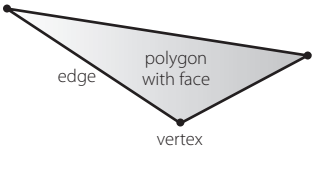


Figure 1.13
Illustration of vertex, edge, polygon, and face

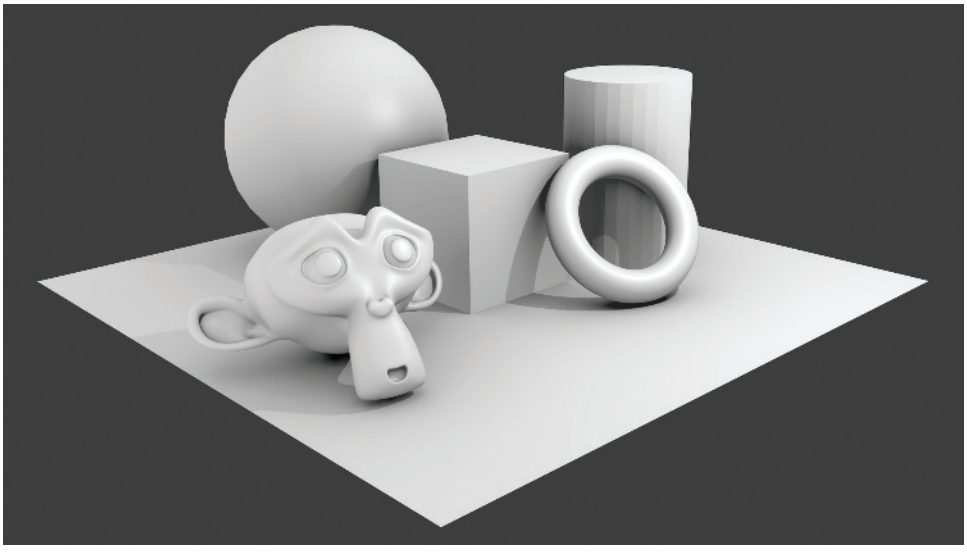


Figure 1.14
Primitives in Blender include cubes, cones, cylinders, and spheres as well as more complex mesh forms.

Setting the Scene

One of the ways meshes are made to look like realistic objects is through the use of *materials.* Materials are sets of directions in 3D programs that tell an object how to simulate the look of real-world substances. Materials can make an object look shiny, matte, opaque, transparent, smooth, rough, and many other types of things. 3D materials consist of a *shader*, a program that describes how a material reacts to light, and *textures*, which will be described in greater detail later in this chapter.

The final elements of a 3D scene are the things that allow users to see their meshes. These are the *lights* and *cameras*. Lights in 3D applications are just like lights in the real world, providing illumination and shadowing to otherwise dull scenes. In many programs, renderings done before lights are added look very flat, whereas in Blender the scene can appear pitch-black. The other element, the camera, simulates the eyes of a viewer looking at the mesh. In Blender, final renders are always done from a camera's point of view. Likewise in games, the camera is the eye of the player in the game world, and its relation to the player character determines whether the game is a first-person, third-person, top-down, side-scrolling, or some other style of game.

Interacting with 3D Models

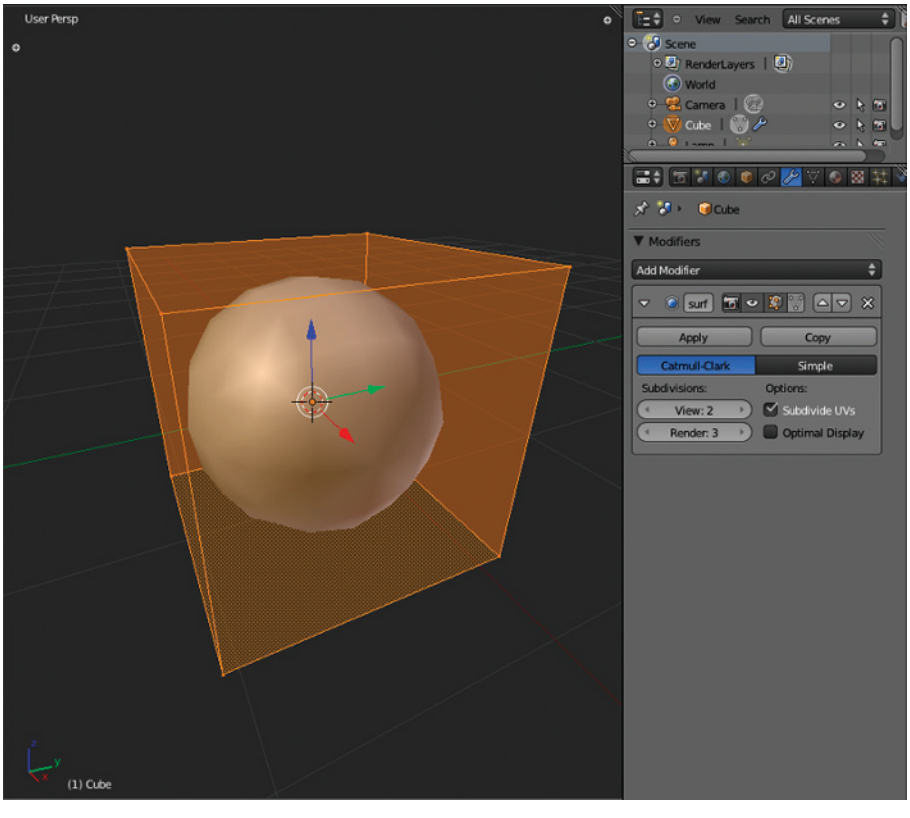
To manipulate meshes and their vertices, edges and faces, lights, and cameras, use their 3D programs' transformation tools. These tools usually take three forms: *translate* (*move*), *rotate,* and *scale*. These tools change an object's position on the xyz coordinate plane. All this is done with simple keyboard and mouse commands that offer a great deal of freedom when creating your 3D forms.

Rendering and Polygon Counts

For film and animation, 3D artists strive for images to appear as close to reality as possible. Even when creating cartoony characters, artists create a realistic look for things like a character's skin, eyes, hair, clothing, and the materials in the character's surroundings. Games are becoming increasingly realistic, but they do not yet have the ability to simulate reality like 3D movies and animations do. This is because the impressive visuals in films are not actually 3D, but instead are 2D rendered images.

When 3D film animators get to work, they model a character in a 3D program like Blender but do so with a focus on visual quality. Often, this requires an artist to apply a *smoothing modifier* to the character, such as Blender's Subdivision Surface modifier, also known as SubSurf for short (Figure 1.15).

Figure 1.15
The Subdivision Surface modifier smoothes a cube into a sphere.



The Subdivision Surface modifier takes the artist's 3D model and smoothes it, creating a polygon surface under the actual surface of the model that has more polygons on it. With more faces come more edges and vertices, which amounts to more data that the computer has to think about.

When everything is said and done, the 3D animator will *render* the image. Rendering is when the computer gathers all of the geometric data of the model as well as material information applied to it and then computes how these things would react in the real world. The output of this process is a final image called a *rendering*. Renderings are 2D images and contain no 3D data. When 3D models are animated, the renders of each frame of animation are stitched together into a movie.

Both rendered images and movies can take hours, days, or even weeks to produce. Some frames of animation in modern movies can take 38 hours to render individually. Studios use *render farms*, banks of linked computers rendering at one time, to compensate for the long hours. Once they are produced, these animations can be stored and viewed easily.

Game artists have the challenge of producing their work for real-time rendered 3D imagery. For every frame of animation, a game's program must compute the 3D models, lights, textures, and interactions between game objects so they can be displayed on the player's screen. To put this in plain English: A video game has to do the rendering tasks of computer animations and then some *as the game is being played!* This means that game artists have to take extra steps to ensure that the game will run smoothly. The first, and perhaps biggest, way that the game artist can accomplish this is by controlling *polygon count*. The polygon count is displayed at the top of the Blender window in an area called the Info Header. The polygon count is indicated in the Info Header with the title "Fa" for faces (Figure 1.16).

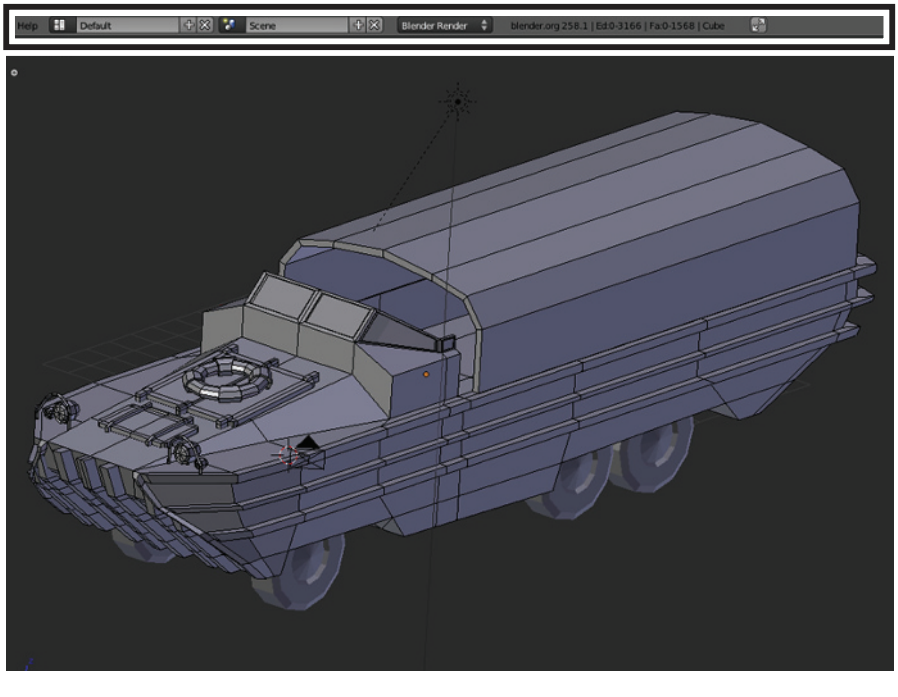


Figure 1.16

The polygon count is typically displayed at the top of the Blender window and shown with the title "Fa" for faces.

Modifiers like Subdivision Surface can increase the number of polygons in a model, so many game designers do not use them.

Video game consoles and engines have what are known as *polygon budgets*, a "highest number" of polygons that can be shown on the screen at one time. These limitations do not mean that your 3D models will look less interesting than those of your 3D animator friends. Quite the contrary—your models will be more efficient.

It is sometimes difficult to gauge how many polygons should go into a model. Often, it depends on three important factors:

- Importance—how important the model is
- Distance—how far the player is from it
- Interactivity—how much the player will interact with it

Consider the player character from Epic Games' *Gears of War*, Marcus Fenix, who clocks in at about 15,000 polygons. He is an important model because he is the player's character in a third-person video game and will need to act in cutscenes. Indeed, a face capable of acting will require a lot of polygons on its own—it could be where over half the polys in even a simple character model are placed. As Marcus is the player's character, he is the most interactive model in the game; he is the player's avatar in the game world. Indeed, he will be viewable very close to the screen and seen closely in cutscenes, so he'd better be his best dressed: 15,000 it is.

A crate or far-away building, on the other hand, can be made for a lot fewer polygons. Even round objects, like an aluminum can or the scope on a rifle, are often made of cylindrical objects with only eight sides. The aforementioned crate can be a simple box solid with six sides; alternatively, it could have some embellishment to give it character. Again, this depends on how far the player is from this object and how much interaction he or she will have with it. The distant building, on the other hand, may be only that six-sided rectangular solid with a texture applied to create the illusion of architectural features. There is no reason to waste polygons on something players will barely see.

The poly counts of monsters and enemies can differ greatly depending on what they are and the three aforementioned factors (importance, distance, and interactivity.) An important boss may have a poly count equal to or greater than that of the player, but the lowest minion may be down in the hundreds or low thousands. If the character appears only at a distance, the poly count may be even lower. All in all, you must weigh many factors when deciding how many polygons to construct enemies with. Simple enemies can be much lower than the player character whereas humanoid ones may be half to two-thirds of their poly count. Remember that when considering how low to make the polygon count on enemies, it is hard to tell the difference in quality when they are running full speed or ducking behind cover.

A fourth factor in how many polygons something can have—a factor so important that it deserves its own paragraph—is the power of the engine and console. *Gears of War* was built on the Unreal engine, which in its various forms has pushed the envelope on visual quality in video games. Additionally, the game is on the Xbox 360, a modern home console. Not every engine or console is as powerful as those for commercial games, however. It is important to know that models like the one for Marcus may be difficult to work with if you are not launching onto similar hardware.

Since Unity is capable of delivering content both to powerful modern consoles and to less powerful mobile devices, it is important to practice using lower poly counts. In earlier model iPhones, for example, a budget of 7,000 visible polygons on the screen was the maximum for optimum performance. This is an incredibly light number compared to the hundreds of thousands possible on consoles. This might require even main characters to be around 1,000 to 2,000 polygons. Link from *The Legend of Zelda: Twilight Princess* on the Wii, on the other hand, is about 6,900 polys and is shown among some very detailed environments. Therefore, the polygon budget for Wii games is much higher than on some mobile platforms, though not as high as high-definition consoles.

Model Topology

In addition to polygon count, it is incredibly important to note that the polygons should be arranged properly on a model so they can animate smoothly. The arrangement of polygons on a 3D model is commonly called *topology*, and is something that many new 3D artists struggle with.

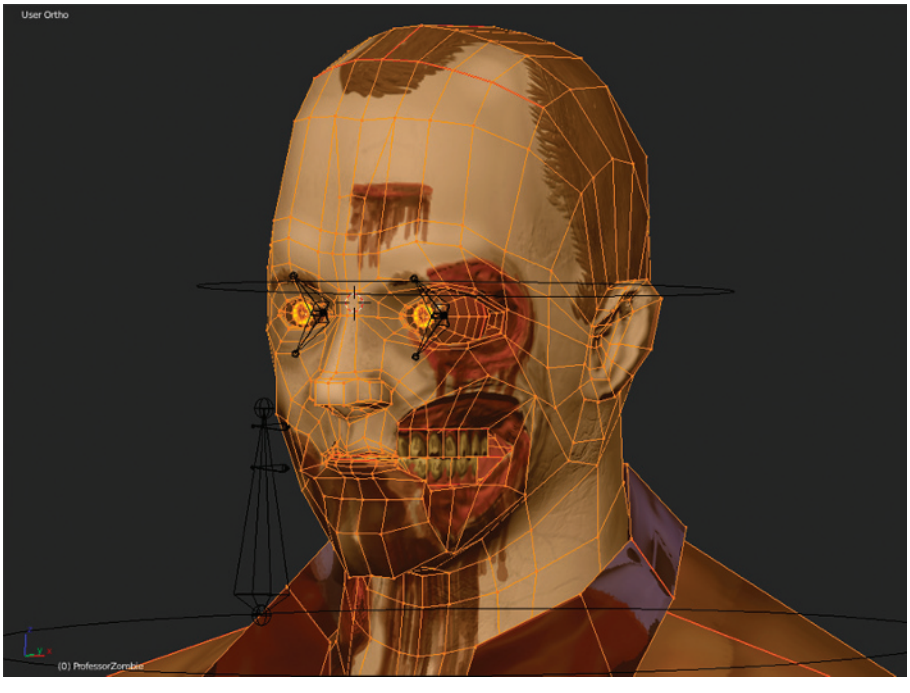
Novices whose background is in other forms of art where the final product is key—especially those who have worked in animation with smoothing modifiers—often model wildly, trying to get the best outcome possible on their characters without giving consideration to topology. When they switch out of smooth mode, they are often shocked to find that their polygons are overlapping one another and entire body parts are awkwardly sticking up through each other. In order for your model to properly function, you must lay out the topology properly. Here are the three most important rules to follow:

- Make sure that all polygons are quads (polygons with four sides).
- Joints on extremities should have at least three edge loops.
- Lay out facial geometry with loops around openings (Figure 1.17).

You'll practice this later in the book, but for now, keep in mind that quads are what 3D animation programs know how to properly distort for animation. Triangles (polygons with three vertices) are possible in Blender but will not deform properly when moving. Any polygon with more than four vertices is known as an ngon, with "n" representing a variable number. You should avoid these altogether when modeling if you want to

animate your mesh. This is simple to avoid in Blender because as of this writing, Blender is incapable of utilizing ngons in the way that other programs can.

Figure 1.17
Character model face highlighting topology



Additionally, know that the number of edge loops in joints and facial features further aids in the animation of a computer model. Having three edge loops at joints allows body parts such as knees and elbows to bend naturally. Likewise, the loops in facial features allow mouth and eye movements to look more natural.

Understanding Normals

Imagine you have created a great 3D model and are ready to bring it into your Unity project, only to find that half of your character's polygons do not show up.

There are two likely reasons that this can happen when you bring a 3D model from Blender into Unity, one of which will be mentioned in Chapter 7, "Rigging for Realistic Movement." The other is that your character's normals may not be facing the right way.

Normals, as applied to 3D modeling, describe the way that a polygon's visible surface is facing. Within Blender's default settings, polygons are not two-sided objects and, in fact, are only visible from one side. If you were to look at a polygonal surface with normals facing inward, for example, you would not be able to see it until you went inside the surface of the model. In Figure 1.18, an image of basic game-level geometry, the exterior portions have normals facing outward, whereas the hallway's normals face inward so players can see the inside of the dungeon.

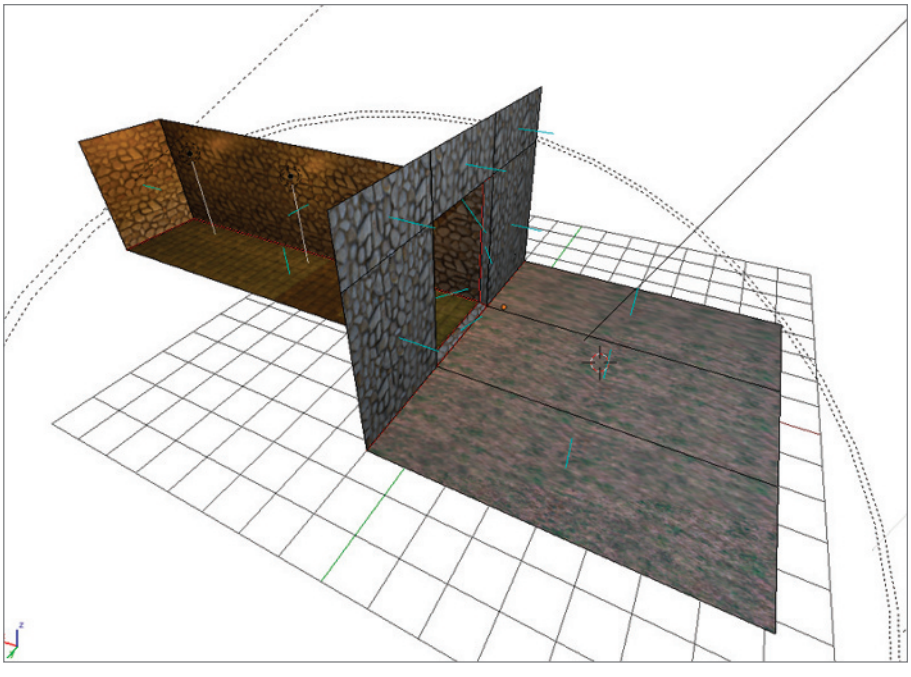


Figure 1.18
In this image of basic game-level geometry, normals are shown with light blue lines.

Remember to watch which way your model's normals are facing as you model—incorrect normals can cause a lot of extra work later on. Blender has a fairly accessible set of functions for both viewing and flipping normals that will be covered in Chapter 3, "Modeling the Character." Likewise, Unity has settings for optimizing normals upon model import; those will be covered in Chapter 10, "Implementing Your Zombie in a Unity Game."

Using Textures Wisely

With all of these concerns over polygon counts and other modeling traps, it is a wonder that video game models look as good as they do. The greater-than-low-poly level of detail you find when playing most games, however, exists because of *textures*.

What are textures? They are image files applied to the surface of a 3D model that change its outward appearance. Textures are one of your most powerful weapons in the struggle against bad game art and graphics. Even the most basic type of texture, the color map, can greatly improve the look of a 3D model. Here are some types of texture maps you will encounter:

Color Maps Also called "diffuse maps," these textures give the 3D model color and definition to features such as body parts and clothing.

Bump Maps As shown in Figure 1.19, these textures are used to make a flat surface appear bumpy. They are typically black and white, with black representing low points in the surface texture and white representing the high points.

Figure 1.19
Bump map

Normal Maps A more powerful version of the bump map, a normal map (Figure 1.20) appears as various RGB values. These textures can make even the lowest-poly model appear high-poly. These textures are created by analyzing the surface of a high-polygon model, such as those created in ZBrush or Blender's sculpting mode, and "baking" them into an image that can be applied as a texture. This type of texture is discussed in-depth in Chapter 5, "Sculpting for Normal Maps."

Figure 1.20
Normal map

Alpha Maps Alpha maps (Figure 1.21) are incredibly useful textures that allow 3D artists to create organic shapes from simple planes by affecting the planes' transparency. In this way, artists can create things like leaves, foliage, chain link fences, dirty windows, and many others with simple geometry. They are also useful for creating particle effects—fountains of plane objects that can imitate smoke, fire, magic, and other special effects. These are often displayed in grayscale, with white areas remaining visible, black areas becoming clear, and everything in between becoming translucent.

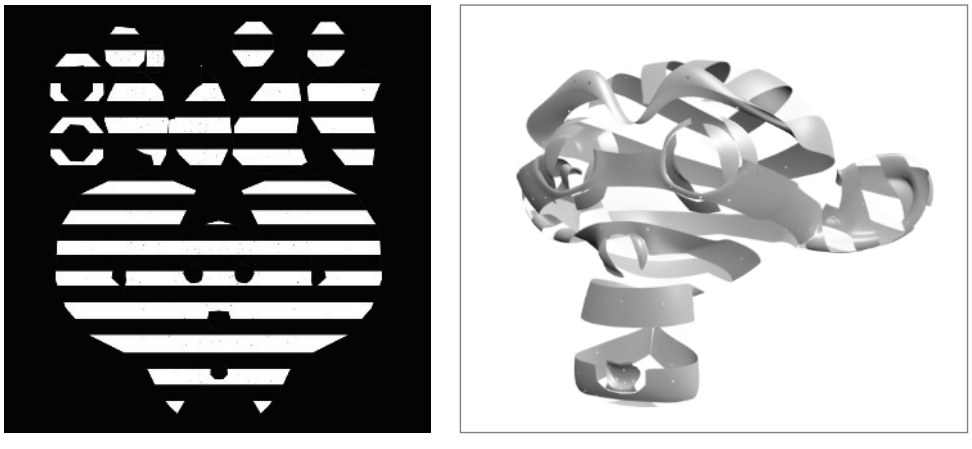


Figure 1.21
Alpha map

Specularity Maps These textures control the specularity, or shininess, of a model's surface. These appear as images having color values ranging from white to black, with white being the shiniest parts and clear being matte-finished areas (Figure 1.22). These are useful in creating the look of worn metal or other materials with varied levels of shininess on their surfaces.

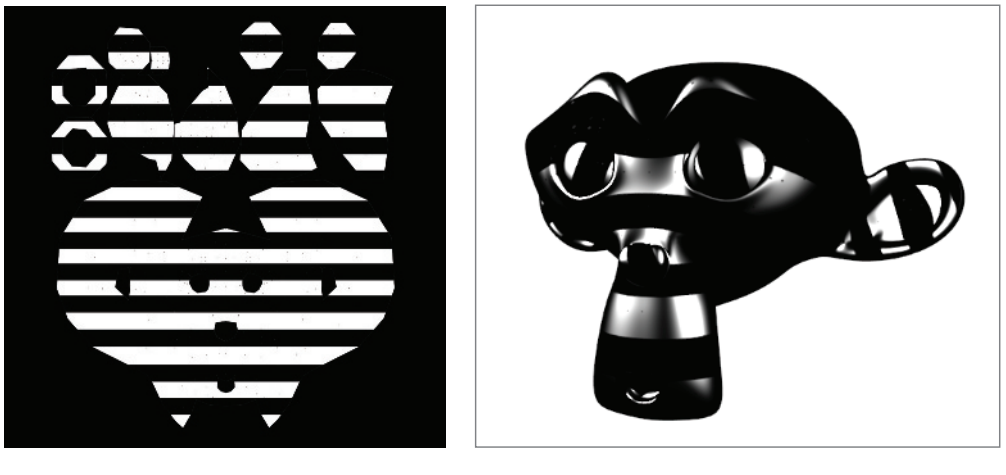


Figure 1.22
The same image used as an alpha map is applied here as a specularity map. Note how the shinier areas are those mapped as white in the texture.

Reflection Maps Also called environment maps, these maps create the illusion that a surface has a mirror-like shininess. The object appears to reflect scenery around it, but in reality it is displaying an image moving across its surface. This image is often a panoramic view of the environment surrounding the object. You often see these used in racing games that depict realistic car models.

Illumination Maps Like specularity maps, these images have a color palette that ranges from white to transparent, with white showing what areas of a model should appear to glow on their own. The most familiar use of these textures is in lit windows for nighttime cityscapes. They can also be useful for creating glowing eyes in powerful characters or enhancing special effects. You will use a map like this on your zombie in Chapter 6, "Digital Painting Color Maps."

Working with Game Engines

Game artists go through all of the steps of preparing 3D models so that the models will eventually work in a game engine—a program that simulates the real world and makes video games function. In this book, you will be utilizing the Unity game engine to make your zombie interactive. However, there is a wide range of game engines available to the do-it-yourself (DIY) game designer.

Game engines are unique among types of software in the gaming industry in that there is no single industry standard that everyone uses. While we will be using Unity in this book, it is useful to understand the scope of engine choices available. The "big name" choices include the Unreal Engine, CryENGINE, and Source Engine, though many studios elect to create their own tool sets. The important thing to remember when choosing and working with a game engine is that it should fit the skills you have and the scale of your project. It does not make sense to give yourself a headache with a 3D engine when your project will look just fine in a 2D engine like Game Editor or FlatRedBall. Game engines vary greatly. Some are more focused on scripting and others are largely WYSIWYG (what you see is what you get) art-based interfaces that show you what your game will look like as you design.

Unity offers the best aspects of many other game engines to the art-focused game designer. It has a WYSIWYG-style interface that lets you drag and drop your art assets into a level. It also has an intuitive importing system that relieves you from many of the headaches that other engines put you through to bring in models. If you are learning about 3D game creation for the first time, Unity is a great choice for learning some of what the game art field involves because it lets you spend more time learning how to work with engines and less time slamming your head into your desk.

When learning about game engines, you must understand the general workflow of bringing artwork into a game and getting it to work, as this is the basis of the game

artist's work with engines. Working with game engines seems overwhelming at first, but if you take to heart the advice in this chapter about polygon count, topology, normals, and the use of textures, you will find that providing art for a game can be relatively painless.

Scripting Happens

Scripting is a topic that many aspiring game artists avoid like a plague but one that is necessary to make great games. Scripting is the act of assigning behaviors to game art assets through a *scripting language*. An engine's scripting language is a type of basic coding that tells the engine how to make an object act while the game program is running.

Scripting is the other half of game art: Once you have an art asset, you need to give the game engine directions so it knows what to do with the asset. A piece of game art, no matter how amazing, is largely useless in a game without a script telling it what to do. It is "all dressed up with nowhere to go."

Many game engines have proprietary scripting languages. By contrast, Unity utilizes three different, widely available languages for scripting: JavaScript, C#, and Boo. JavaScript is a powerful language primarily used for adding functionality to websites. It is also "object oriented," which means that elements of code are organized according to the object utilizing them. This makes object-oriented languages simple to understand for newcomers. C#, another object-oriented language, is designed to operate on Microsoft's .NET Framework, a framework for developing software in Windows. Boo is a derivative of the open-source language Python, which itself emphasizes code readability. Python is also the scripting language of Blender, which makes knowing Python and the languages it has inspired a versatile skill.

If this sounds a bit over your head, fear not! When you encounter scripting in Chapter 10, you will not be alone. This book will walk you through several useful scripts and how to implement them in the Unity engine, which will put the interactive icing on the cake for your game character project.

So, with all of your newfound game art knowledge, O intrepid DIY game designer, let us venture onward into the frontier of 3D modeling and learn the basics of Blender, the open-source 3D art environment!