Game Engine Architecture

Chapter 11

The Rendering Engine

Foundations of Depth-Buffered Triangle Rasterization

- Rendering a three-dimensional scene involves the following basic steps:
  - A *virtual scene* is described, usually in terms of 3D surfaces represented in some mathematical form.
  - A *virtual camera* is positioned and oriented to produce the desired view of the scene. Typically the camera is modeled as an idealized focal point, with an imaging surface hovering some small distance in front of it, composed of *virtual light sensors* corresponding to the picture elements (*pixels*) of the target display device.
  - Various *light sources* are defined. These sources provide all the light rays that will interact with and reflect off the objects in the environment and eventually find their way onto the image-sensing surface of the virtual camera.
  - The *visual properties* of the surfaces in the scene are described. This defines how light should interact with each surface.
  - For each pixel within the imaging rectangle, the rendering engine calculates the color and intensity of the light ray(s) converging on the virtual camera's focal point through that pixel. This is known as *solving the rendering equation* (also called the *shading equation*).

Describing a Scene

- A real-world scene is composed of objects
  - *Opaque*
  - *Transparent*
  - *Translucent*
- Opaque objects can be rendered by considering only their *surfaces*
- When rendering a transparent or translucent object, we really should model how light is reflected, refracted, scattered and absorbed
- Game engines render the surfaces of transparent and translucent objects in almost the same way opaque objects are rendered.
  - *alpha* is used to describe how opaque or transparent a surface is.
- most game rendering engines are primarily concerned with rendering *surfaces*.

Representations Used by High-End Rendering Packages

- A surface is a two-dimensional sheet comprised of an infinite number of points in three-dimensional space
- Some surfaces can be described exactly in analytical form using a *parametric surface equation*.
- In the film industry, surfaces are often represented by a collection of rectangular *patches* each formed from a two-dimensional spline defined by a small number of control points.
- Various kinds of splines are used, including Bézier surfaces (e.g., bicubic patches, which are third-order Béziers), nonuniform rational B-splines (NURBS), Bézier triangles and *N*-patches (also known as *normal patches)*
- High-end film rendering engines like Pixar's RenderMan use *subdivision surfaces* to define geometric shapes (the Catmull-Clark algorithm).

Triangle Meshes

- Triangles are the polygon of choice for real-time rendering engine:
  - *The triangle is the simplest type of polygon*
  - *A triangle is always planar*
  - *Triangles remain triangles under most kinds of transformations, including affine transforms and perspective projections*
  - *Virtually all commercial graphics-acceleration hardware is designed around triangle rasterization*

Tessellation

- Dividing a surface up into a collection of discrete polygons
- *Triangulation* is tessellation of a surface into triangles.
- Fixed tessellation can cause an object's *silhouette edges* to look blocky
- Surfaces can be tessellated based on distance from the camera, so that every triangle is less than one pixel in size.
- The first LOD, often called LOD 0, represents the highest level of tessellation; it is used when the object is very close to the camera. Subsequent LODs are tessellated at lower and lower resolutions

dynamic tessellation

- Some game engines apply *dynamic tessellation* techniques to expansive meshes like water or terrain.
- *Progressive meshes:*
  - a single high-resolution mesh is created for display when the object is very close to the camera.
  - This mesh is automatically detessellated as the object gets farther away by collapsing certain edges

Mesh Instancing

- Any one mesh might appear many times in a scene
- We call each such object a *mesh instance*.
- A mesh instance contains a reference to its shared mesh data and also includes a transformation matrix (*model-to-world matrix)*.

Color Spaces and Color Models

- A *color model* is a three-dimensional coordinate system that measures colors.
- A *color space* is a specific standard for how numerical colors in a particular color model should be mapped onto the colors perceived by human beings in the real world.
- The most commonly used color model in computer graphics is the RGB model.
- In this model, color space is represented by a unit cube, with the relative intensities of red, green and blue light measured along its axes.
- The red, green and blue components are called *color channels*.
- In the canonical RGB color model, each channel ranges from zero to one. So the color (0, 0, 0) represents black, while (1, 1,1)
- A color format is defined by the number of *bits perpixel it occupies*.
- The RGB888 format uses eight bits per channel, for a total of 24 bits per pixel. In this format, each channel ranges from 0 to 255 rather than [0,1]
- A fourth channel called *alpha* is often tacked on to RGB color vectors.

Vertex Attributes

- A typical triangle mesh includes some or all of the following attributes at each vertex
  - *Position vector*
  - *Vertex normal*
  - *Vertex tangent*
  - *bitangent*
  - *Diffuse color*
  - *Specular color*
  - *Texture coordinates*
  - *Skinning weights*
- *Vertex tangent* **t***i* and *bitangent* **b***i* are two unit vectors that lie perpendicular to one another and to the vertex normal **n***i*. Together
- the three vectors **n***i*, **t***i* and **b***i* define a set of coordinate axes known as *tangent space*.
- This space is used for various per-pixel lighting calculations, such as normal mapping and environment mapping.

Vertex Formats

```
// Simplest possible vertex -- position only (useful for
// shadow volume extrusion, silhouette edge detection
// for cartoon rendering, z-prepass, etc.)
struct Vertex1P
{
Vector3 m_p; // position
};
// A typical vertex format with position, vertex normal
// and one set of texture coordinates.
struct Vertex1P1N1UV
{
Vector3 m_p; // position
Vector3 m_n; // vertex normal
F32 m_uv[2]; // (u, v) texture coordinate
};
// A skinned vertex with position, diffuse and specular
// colors and four weighted joint influences.
struct Vertex1P1D1S2UV4J
{
Vector3 m_p; // position
Color4 m_d; // diffuse color and translucency
Color4 m_S; // specular color
F32 m_uv0[2]; // first set of tex coords
F32 m_uv1[2]; // second set of tex coords
U8 m_k[4]; // four joint indices, and...
F32 m_w[3]; // three joint weights, for skinning (fourth is calc'd from the first three)
};
```

Vertex Formats

- the number of permutations of vertex attributes—and therefore the number of distinct vertex formats—can grow to be extremely large.
- Management of all these vertex formats is a common source of headaches for any graphics programmer.
- In practical graphics applications, many of the theoretically possible vertex formats are simply not useful, or they cannot be handled by the graphics hardware or the game's shaders.
- Some game teams also limit themselves to a subset of the useful/feasible vertex formats in order to keep things more manageable.
- For example, they might only allow zero, two or four joint weights per vertex, or they might decide to support no more than two sets of texture coordinates per vertex.
- Some GPUs are capable of extracting a subset of attributes from a vertex data structure, so game teams can also choose to use a single "überformat" for all meshes and let the hardware select the relevant attributes based on the requirements of the shader.

Types of Textures

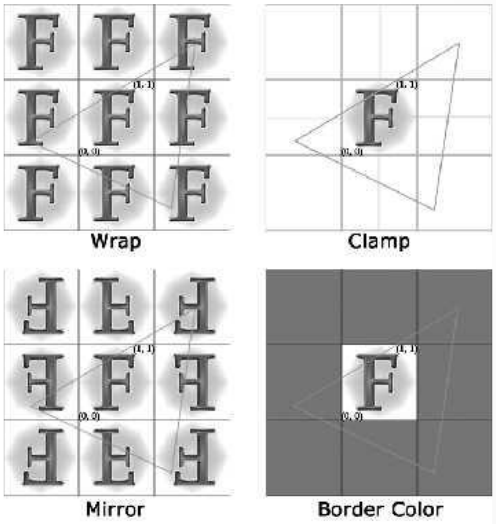
- The most common type of texture is known as a *diffuse map*, or *albedo map*.
- It describes the diffuse surface color at each texel on a surface and acts like a decal or paint job on the surface.
- Other types of textures including:
  - *normal maps* (which store unit normal vectors at each texel, encoded as RGB values)
  - *gloss maps* (which encode how shiny a surface should be at each texel),
  - environment maps (which contain a picture of the surrounding environment for rendering reflections)
- We can actually use texture maps to store any information that we happen to need in our lighting calculations.
  - For example, a one-dimensional texture could be used to store sampled values of a complex math function, a color-to-color mapping table, or any other kind of look-up table (LUT).

Texture Coordinates

- A texture coordinate is usually represented by a normalized pair of numbers denoted (*u*, *v*).
- These coordinates always range from (0, 0) at the bottom left corner of the texture to (1, 1) at the top right.
- Using normalized coordinates like this allows the same coordinate system to be used regardless of the dimensions of the texture.
- To map a triangle onto a 2D texture, we simply specify a pair of texture coordinates (*ui*, *vi*) at each vertex *i*.
- This effectively maps the triangle onto the image plane in texture space.

Texture Addressing Modes

- Texture coordinates are permitted to extend beyond the [0, 1] range.
- The graphics hardware can handle out-of-range texture coordinates in any one of the following ways.
- These are known as *texture addressing modes*
- which mode is used is under the control of the user.



Texture Formats

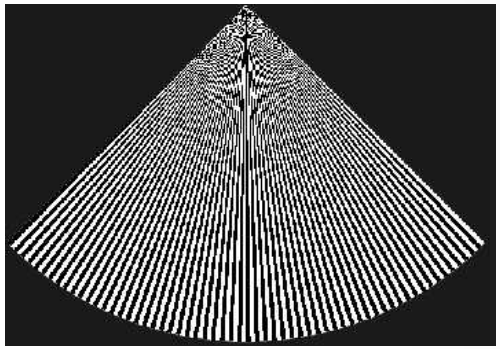
- Texture bitmaps can be stored on disk in virtually any image format, provided
- your game engine includes the code necessary to read it into memory.
  - Common formats include Targa (.tga),
  - Portable Network Graphics (.png),
  - Windows Bitmap (.bmp) and
  - Tagged Image File Format (.tif).
- In memory, textures are usually represented as two-dimensional (strided) arrays of pixels using various
- color formats, including RGB888, RGBA8888, RGB565, RGBA5551 and so on.
- Most modern graphics cards and graphics APIs support *compressed textures*.
  - DirectX supports a family of compressed formats known as DXT or S3 Texture Compression (S3TC).

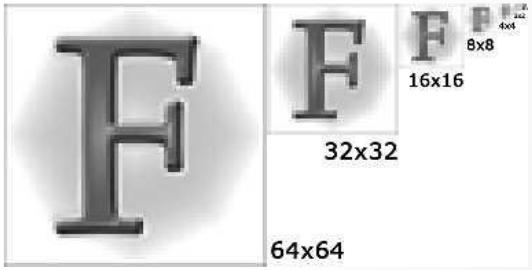
Texel Density

- Imagine rendering a full-screen quad (a rectangle composed of two triangles) that has been mapped with a texture whose resolution exactly matches that of the screen.
- In this case, each texel maps exactly to a single pixel on-screen, and we say that the *texel density* (ratio of texels to pixels) is one.
- texel density is not a fixed quantity—it changes as a texture-mapped object moves relative to the camera.

Mipmapping

- A texel density greater than one can lead to a moiré pattern
- Ideally we'd like to maintain a texel density that is close to one at all times, for both nearby and distant objects.
- This is impossible to achieve exactly, but it can be approximated via a technique called *mipmapping*.
- For each texture, we create a sequence of lower-resolution bitmaps, each of which is one-half the width and one-half the height of its predecessor.
- We call each of these images a *mipmap*, or *mip level*.





World-Space Texel Density

- The term "texel density" can also be used to describe the ratio of texels to world-space area on a textured surface.
- For example, a 2 m cube mapped with a 256 X 256 texture:
  - texel density = $256^2/2^2 = 16{,}384$.
- We will call this *world-space texel density* to differentiate it from screen-space texel density
- World-space texel density need not be close to one, and in fact the specific value will usually be much greater than one and depends entirely upon your
- choice of world units.
- Many game studios provide their art teams with guidelines and in-engine texel density visualization tools in an effort to ensure that all objects in the game have a reasonably consistent world-space texel density.

Texture Filtering

- When rendering a pixel of a textured triangle, the graphics hardware samples the texture map by considering where the pixel center falls in texture space.
- There is usually not a clean one-to-one mapping between texels and pixels
- Therefore, the graphics hardware usually has to sample more than one texel and blend the resulting colors to arrive at the actual sampled texel color.
- We call this *texture filtering*.

Texture Filtering

- *Nearest neighbor:* In this crude approach, the texel whose center is closest to the pixel center is selected. When mipmapping is enabled, the mip level is selected whose resolution is nearest to but greater than the ideal theoretical resolution needed to achieve a screen-space texel density of one.
- *Bilinear:* the four texels surrounding the pixel center are sampled, and the resulting color is a weighted average of their colors. When mipmapping is enabled, the nearest mip level is selected.
- *Trilinear:* bilinear filtering is used on each of the two nearest mip levels (one higher-res than the ideal and the other lower-res), and these results are then linearly interpolated.
- *Anisotropic:* Both bilinear and trilinear filtering sample 2 X 2 square blocks of texels. This is the right thing to do when the textured surface is being viewed head-on, but it's incorrect when the surface is at an oblique angle relative to the virtual screen plane. Anisotropic filtering samples texels within a trapezoidal region corresponding to the view angle, thereby increasing the quality of textured surfaces when viewed at an angle.

Materials

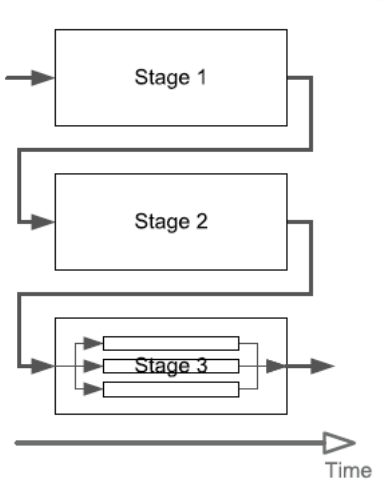
- A *material* is a complete description of the visual properties of a mesh.
- Includes:
  - a specification of the textures that are mapped to its surface
  - which shader programs to use when rendering the mesh
- vertex attributes are not considered to be part of the material.
- mesh-material pair contains all the information we need to render the object.
  - sometimes called *render packets*

Submeshes

- A 3D model typically uses more than one material.
- For example, a model of a human would have separate materials for the hair, skin, eyes, teeth and various kinds of clothing.
- A mesh is usually divided into *submeshes*,
- each submesh is mapped to a single material.
- The OGRE rendering engine implements this design via its Ogre::SubMesh class

The Rendering Pipeline

- In real-time game rendering engines, the high-level rendering steps are implemented using a software/hardware architecture known as a *pipeline*.
- A pipeline is just an ordered chain of computational stages, each with a specific purpose, operating on a stream of input data items and producing a stream of output data.
- Each stage of a pipeline can typically operate independently of the other stages.
- The stages all operate in parallel, and some stages are capable of operating on multiple data items simultaneously as well.



Overview of the Rendering Pipeline

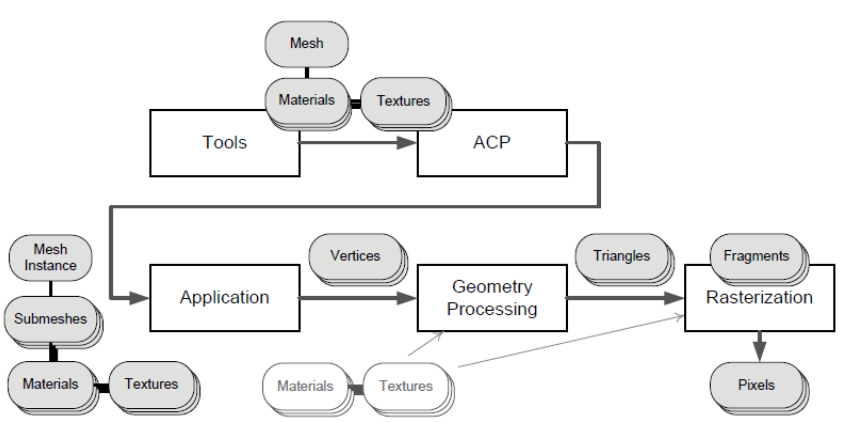
- *Tools stage (offline)*. Geometry and surface properties (materials) are defined.

- *Asset conditioning stage (offline)*. The geometry and material data are processed by the asset conditioning pipeline (ACP) into an engine-ready format.

- *Application stage (CPU)*. Potentially visible mesh instances are identified and submitted to the graphics hardware along with their materials for rendering.

- *Geometry processing stage (GPU)*. Vertices are transformed and lit and projected into homogeneous clip space. Triangles are processed by the optional geometry shader and then clipped to the frustum.

- *Rasterization stage (GPU)*. Triangles are converted into fragments that are shaded, passed through various tests (z-test, alpha test, stencil test, etc.) and finally blended into the frame



The Tools Stage

- In the tools stage, meshes are authored by 3D modelers in a *digital content creation* (DCC) application like Maya, 3ds Max, Lightwave, Softimage/XSI, SketchUp, etc.
- The models may be defined using any convenient surface description—NURBS, quads, triangles, etc.
- Models are invariably tessellated into triangles prior to rendering by the runtime portion of the pipeline.
- The vertices of a mesh may also be skinned. This involves associating each vertex with one or more joints in an articulated skeletal structure, along with weights describing each joint's relative influence over the vertex.
- Materials are also defined by the artists during the tools stage. This involves selecting a shader for each material, selecting textures as required by the shader, and specifying the configuration parameters and options of each shader.

Material Editor

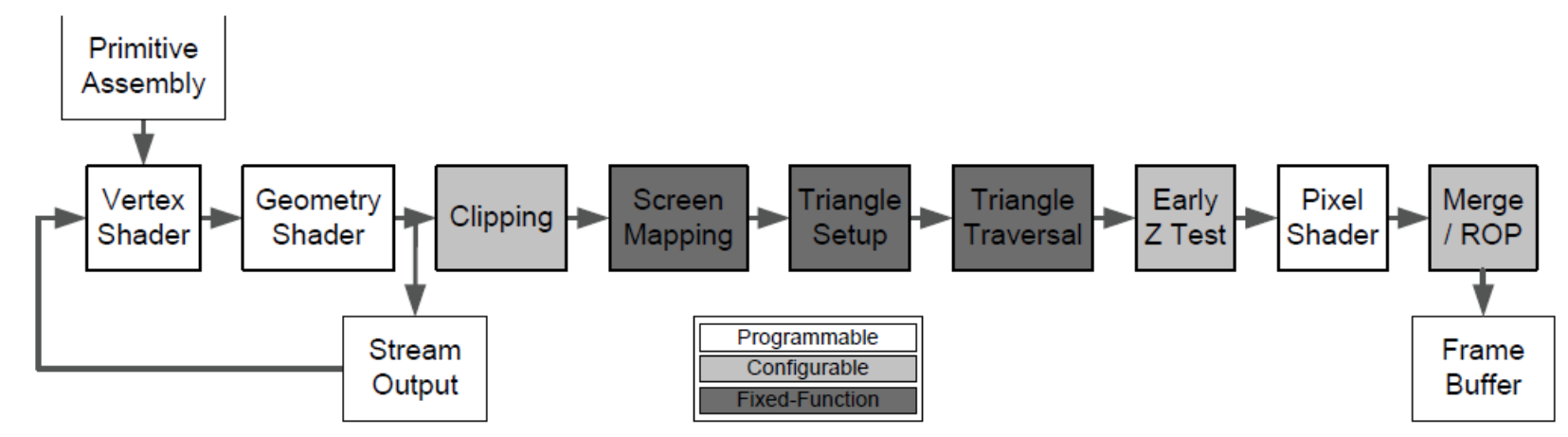
- The material editor is sometimes integrated directly into the DCC application as a plug-in, or it may be a stand-alone program
- NVIDIA is no longer updating Fx Composer, and it only supports shader models up to DirectX 10.
- they do offer a new Visual Studio plugin called NVIDIA® Nsight™ Visual Studio Edition
- NVIDIA Nsight Visual Studio Edition | NVIDIA Developer
- Nsight provides powerful shader authoring and debugging facilities.
- The Unreal Engine also provides a graphical shader editor called Material Editor;
- Game teams build up a library of materials from which to choose, and the individual meshes refer to the materials in a loosely coupled manner.

The Asset Conditioning Stage

- The asset conditioning stage is itself a pipeline, sometimes called the *asset conditioning pipeline* (ACP) or the *tools pipeline*
- its job is to export, process and link together multiple types of assets into a cohesive whole.
- For example, a 3D model is comprised of geometry (vertex and index buffers), materials, textures and an optional skeleton.
- The ACP ensures that all of the individual assets referenced by a 3D model are available and ready to be loaded by the engine.
- Geometric and material data is extracted from the DCC application and is usually stored in a platform-independent intermediate format. The data is then further processed into one or more platform-specific formats, depending on how many target platforms the engine supports.
- For example, mesh data targeted for the Xbox One or PS4 might be output as index and vertex buffers that are ready to be consumed by the GPU; on the PS3, geometry might be produced in compressed data streams that are ready to be DMA'd to the SPUs for decompression.

The GPU Pipeline

- Virtually all GPUs break the graphics pipeline into the substages depicted in Figure

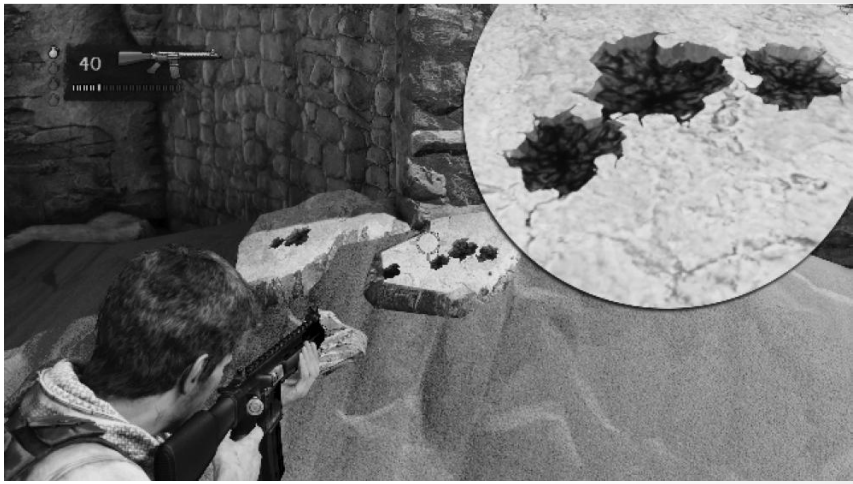


Particle Effects

- A particle rendering system is concerned with rendering amorphous objects like clouds of smoke, sparks, flame and so on. These are called *particle effects*.
- It is composed of a very *large number* of *relatively simple* pieces of geometry— most often simple cards called *quads*, composed of two triangles each.
- The geometry is often *camera-facing* (i.e., billboarded), meaning that the engine must take steps to ensure that the face normals of each quad always point directly at the camera's focal point.
- Its materials are almost always *semitransparent* or *translucent*. As such, particle effects have some stringent *rendering order* constraints that do not apply to the majority of opaque objects in a scene.
- Particles *animate* in a rich variety of ways. Their positions, orientations, sizes (scales), texture coordinates and many of their shader parameters vary from frame to frame. These changes are defined either by handauthored animation curves or via procedural methods.
- Particles are typically *spawned and killed* continually. A particle emitter is a logical entity in the world that creates particles at some user-specified rate; particles are killed when they hit a predefined death plane, or when they have lived for a user-defined length of time, or as decided by some other user-specified criteria.

Decals

- A *decal* is a relatively small piece of geometry that is overlaid on top of the regular geometry in the scene, allowing the visual appearance of the surface to be modified dynamically.
  - Examples include bullet holes, foot prints, scratches, cracks, etc.
- Modern engines models a decal as a rectangular area that is to be projected along a ray into the scene (a rectangular prism in 3D space)
- Whatever surface the prism intersects first becomes the surface of the decal. The triangles of the intersected geometry are extracted and clipped against the four bounding planes of the decal's projected prism.
- The resulting triangles are texture-mapped with a desired decal texture by generating appropriate texture coordinates for each vertex.
- These texture mapped triangles are then rendered over the top of the regular scene, often using parallax mapping to give them the illusion of depth and with a slight z-bias.

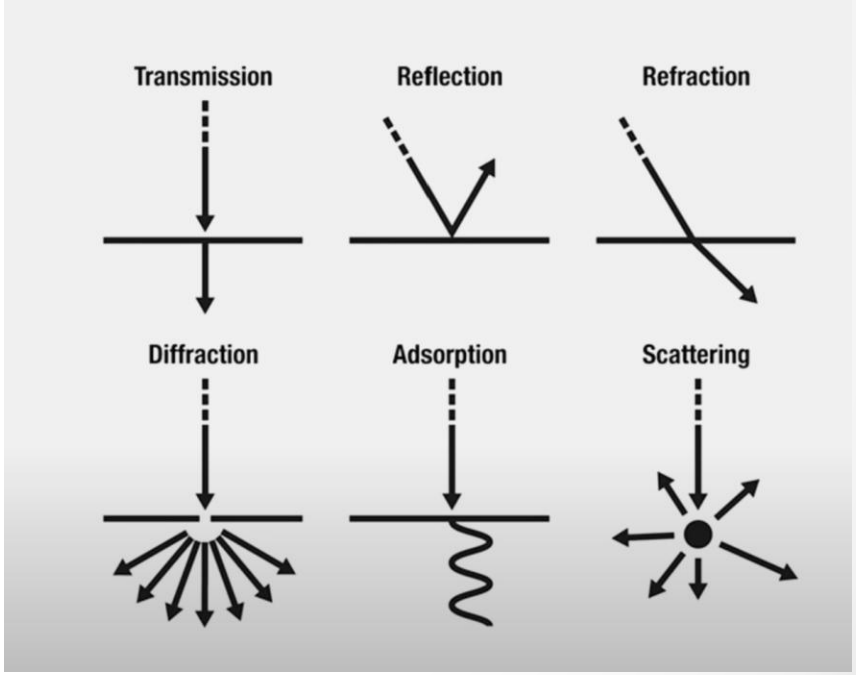


Environmental Effects

- Skies: The sky in a game world needs to contain vivid detail.
- One simple approach is to fill the frame buffer with the sky texture prior to rendering any 3D geometry.
- The sky texture should be rendered at an approximate 1:1 texel-to-pixel ratio so that the texture is roughly or exactly the resolution of the screen.
- The sky texture can be rotated and scrolled to correspond to the motions of the camera in-game.
- During rendering of the sky, we make sure to set the depth of all pixels in the frame buffer to the maximum possible depth value. This ensures that the 3D scene elements will always sort on top of the sky.
- On modern game platforms, where pixel shading costs can be high, sky rendering is often done *after* the rest of the scene has been rendered. First the z-buffer is cleared to the maximum z-value. Then the scene is rendered.
- Finally the sky is rendered, with z-testing enabled, z writing turned off, and using a z-test value that is one less than the maximum. This causes the sky to be drawn only where it is not occluded by closer objects like terrain, buildings and trees.

Lighting

- Light can have many complex interactions with matter.
- It can be absorbed
- It can be reflected.
- It can be transmitted.
- It can be refracted (bent) in the process.
- It can be diffracted when passing through very narrow openings.



Baking vs. Progressive Lightmapper in Unity

Often you want to calculate lighting beforehand both to get more detailed lighting but also to save on performance this is called baking.

When you bake your lighting, unity calculates all the lighting values and turns them into large textures that are then overlaid on top of your objects.

The progressive light mapper is a path tracing based light mapper which means

that it will realistically simulate lighting bouncing around the scene.

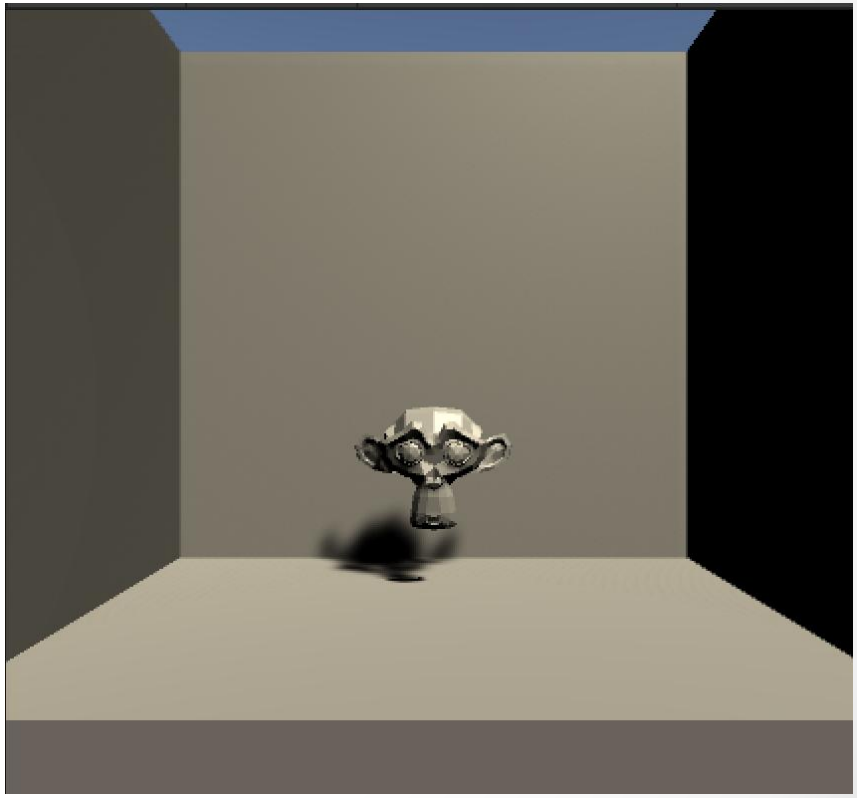
It does this by gradually sending out light rays from the camera towards the light sources.

This means that you can see the result as it renders in the editor and you can adjust light properties without having to re-render the entire scene.

Lab1: Create an indoor right

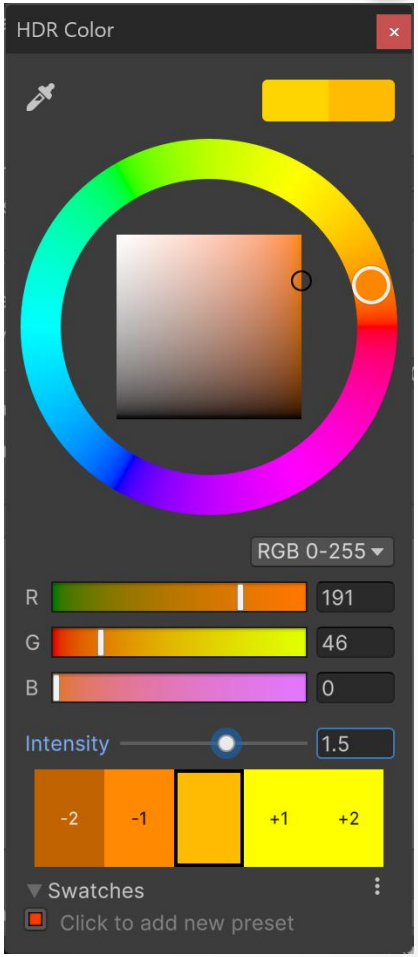
- Create a new 3D pipeline project
- Import the Starter file in the Github: GAME3121\Week13\LightStarterDemo1.UnityPackage
  - Monkey(from Blender) is in the walls
- Remove the directional Light from the hierarchy

What we want to do is set up a few light sources. By default, every unity scene comes with a directional light go ahead and remove this that's because I want this to be an indoor scene so instead let's create our own lights of course we could go right click light and select one of the predefined lights or we could simply create an object with an emissive material and that's what we're going to be doing here!



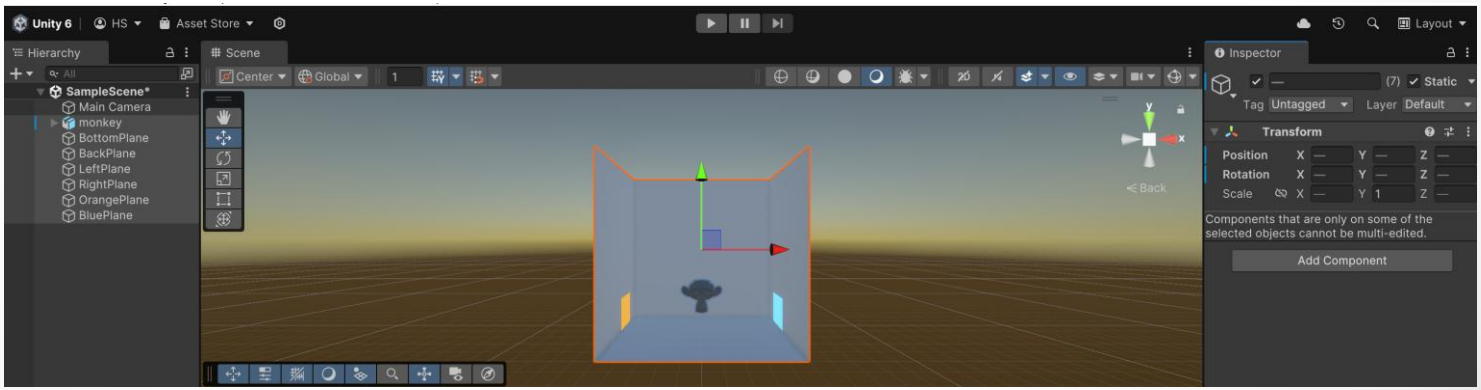
Create an emmisive light

- Create a new Plane:OrangePlane (Rotation(0,0,-90), Scale(0.2,1,0.2), Tranformation(-4.99,1,0))
- Create a new Plane:BluePlane (Rotation(0,-180,-90), Scale(0.2,1,0.2), Tranformation(4.99,1,0))
- Create a new Material: OrangeLight, Checkbox the "Emmission" and change the HDR color to Orange #FFA400, The amount of Intensity:1.5, Change the global illumination to Baked
- Create a new Material: BlueLight, Checkbox the "Emmission" and change the HDR color to Orange #00CAFF, The amount of Intensity 2, Change the global illumination to Baked.



Static

- We must make sure that our light is static. To do that, select all the items in the hierarchy (except Main Camera) and check the static box in the inspector.



Generate Light

- Go to Window --> Rendering --> Lighting
- On Environment Tab:

  a) Set the skybox material and sun source to "None" (this is not outdoor scene any more)

  b) Change the ambient color to black (we don't need any light from outside)
- On Scene Tab:

  a) Uncheck the realtime global illumination

  b) change the lightmapper to "Progressive GPU"
- Generate Lighting