Game Development Training

Introduction to Game Development

Unity 3D Training

Getting Started with Unreal Engine 4

Introduction to Game Development – 2

The Development Flow

- Part 1 : Rendering and Graphics
- Part 2 : Physics
- Part 3 : Animation System
- Part 4 : Navigation and Pathfinding
- Part 5 : The Development Pipeline

Introduction to Game Development – 2

The Development Flow

Part 1 : Rendering and Graphics

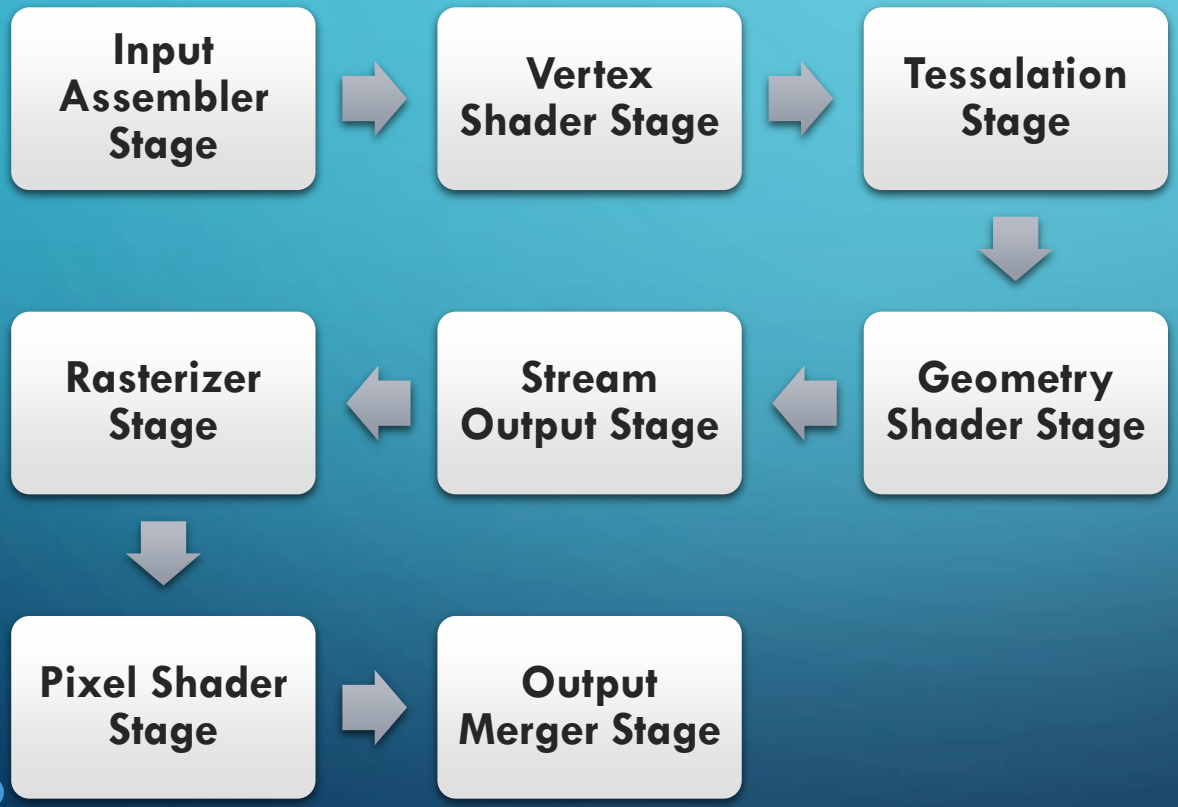
DirectX and OpenGL

Microsoft DirectX is the graphics technology powering today's most impressive games. It is a collection of application programming interfaces(APIs) for handling task related to enhanced graphics , especially in game programming and videos, on Microsoft platforms, like Windows and Xbox. Direct3D is the graphics API and is a part of DirectX.

Open Graphics Library(OpenGL) is a cross platform API for rendering 2D and 3D vector graphics.

DirecX Rendering Pipeline

The Rendering or Graphics Pipeline refers to the sequence of steps used to create 2D representation of a 3D scene.

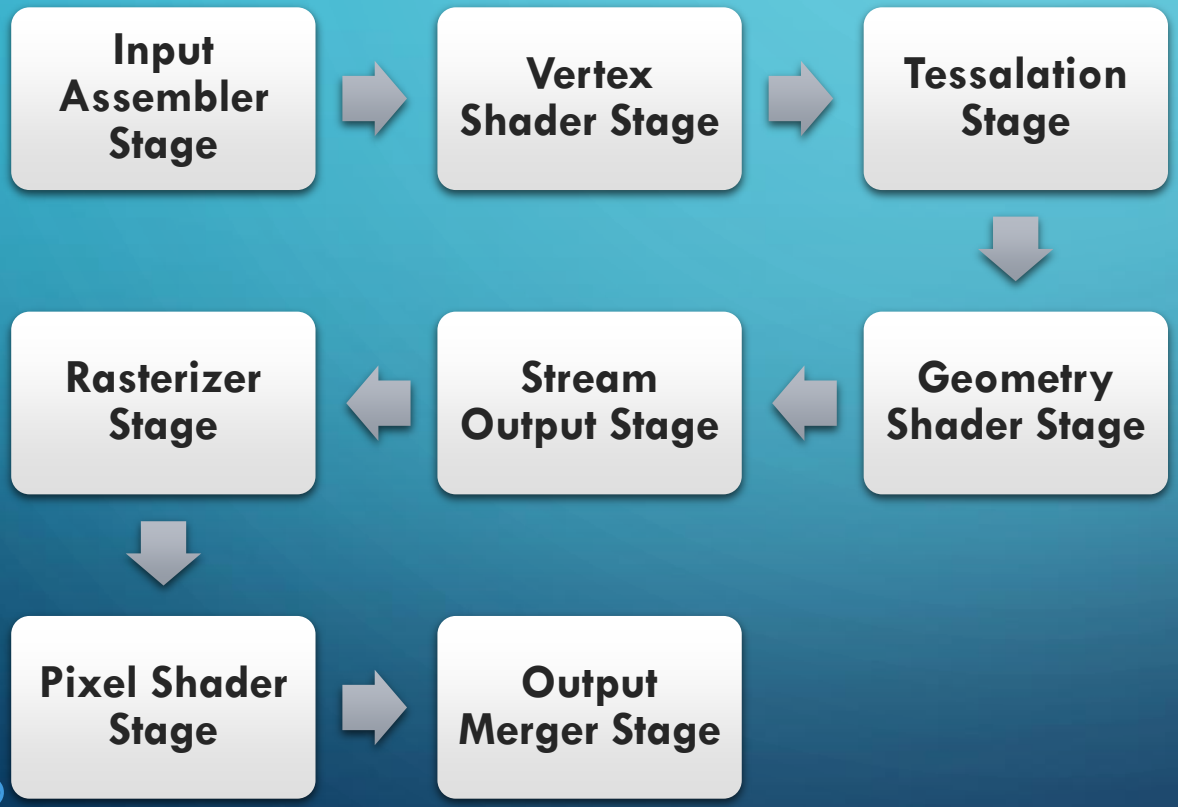


Input Assembler Stage

The purpose of the input-assembler stage is to read primitive data (points, lines and/or triangles) from user-filled buffers and assemble the data into primitives that will be used by the other pipeline stages.

DirecX Rendering Pipeline

The Rendering or Graphics Pipeline refers to the sequence of steps used to create 2D representation of a 3D scene.

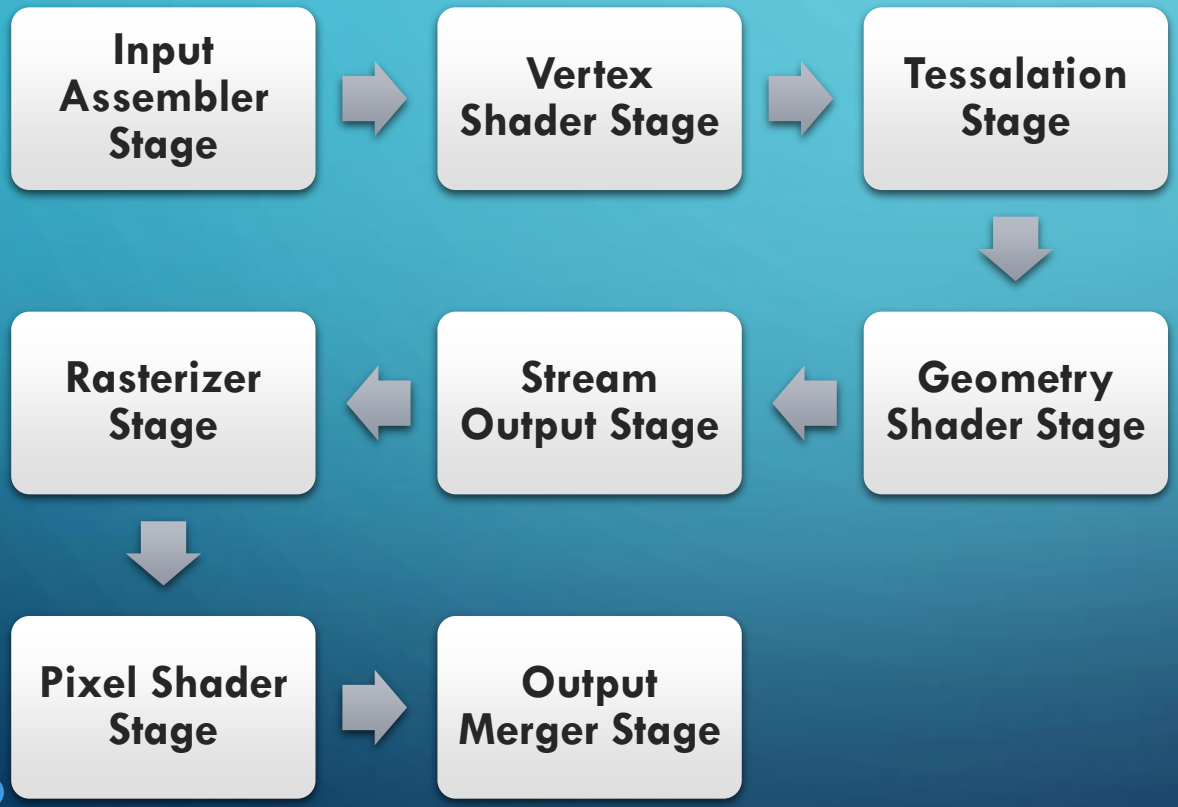


Vertex Shader Stage

The vertex-shader (VS) stage processes vertices from the input assembler, performing per-vertex operations such as transformations, morphing, and per-vertex lighting. Vertex shaders always operate on a single input vertex and produce a single output vertex.

DirecX Rendering Pipeline

The Rendering or Graphics Pipeline refers to the sequence of steps used to create 2D representation of a 3D scene.



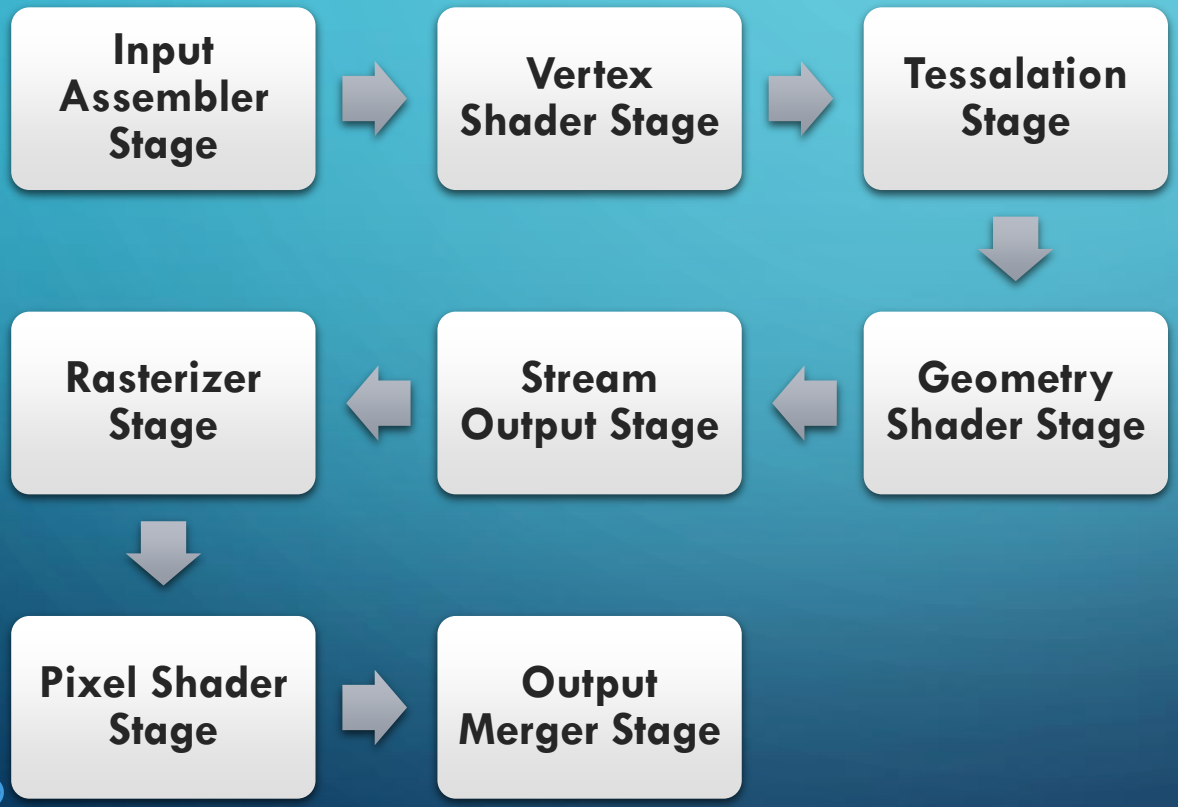
Tessellation Stages

Tessellation converts low-detail subdivision surfaces into higher-detail primitives on the GPU.

By implementing tessellation in hardware, a graphics pipeline can evaluate lower detail (lower polygon count) models and render in higher detail.

DirecX Rendering Pipeline

The Rendering or Graphics Pipeline refers to the sequence of steps used to create 2D representation of a 3D scene.

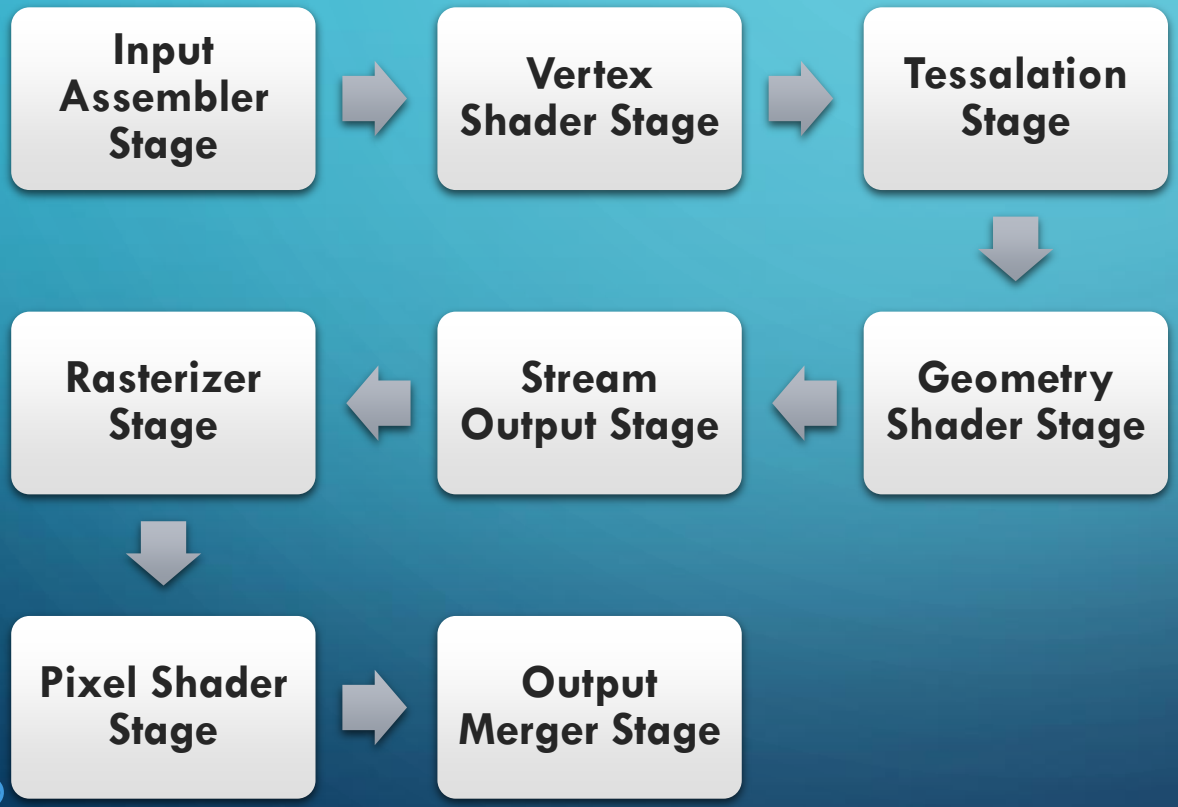


Geometry Shader Stage

The geometry-shader (GS) stage runs application-specified shader code with vertices as input and the ability to generate vertices on output.

DirecX Rendering Pipeline

The Rendering or Graphics Pipeline refers to the sequence of steps used to create 2D representation of a 3D scene.

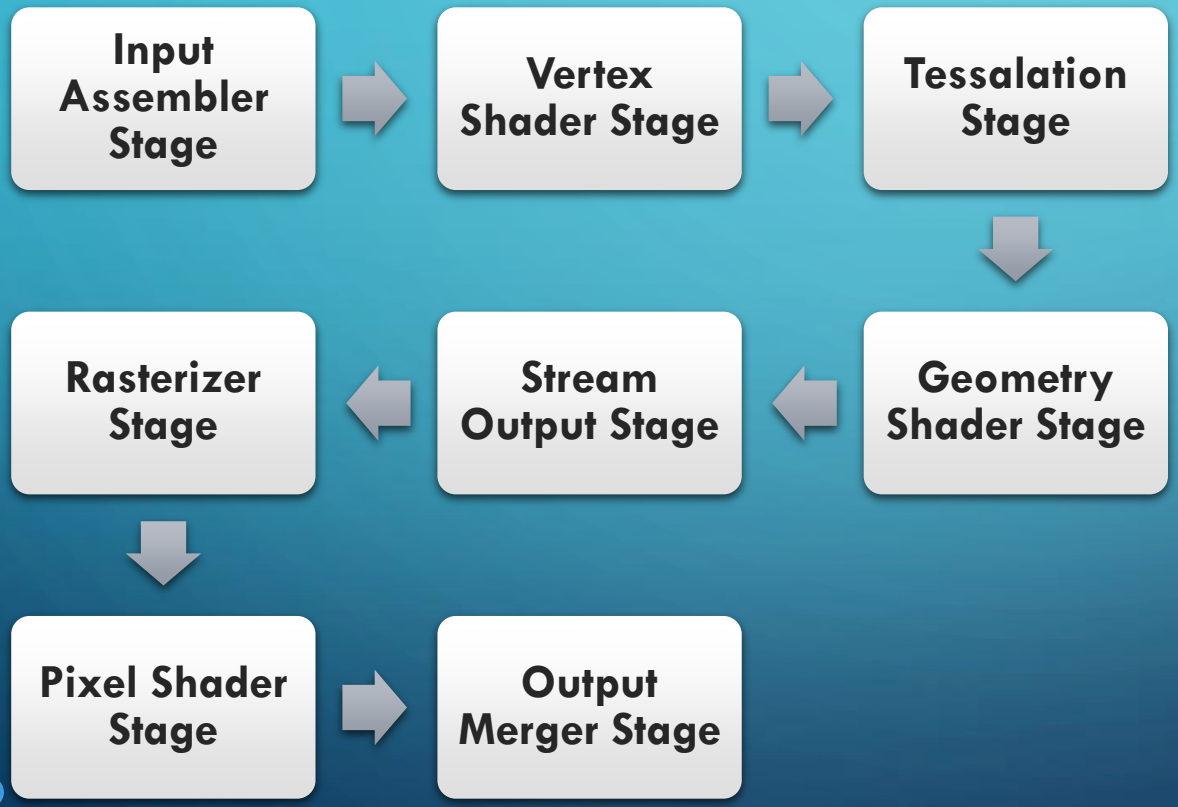


Stream Output Stage

The purpose of the stream-output stage is to continuously output (or stream) vertex data from the geometry-shader stage to one or more buffers in memory

DirecX Rendering Pipeline

The Rendering or Graphics Pipeline refers to the sequence of steps used to create 2D representation of a 3D scene.

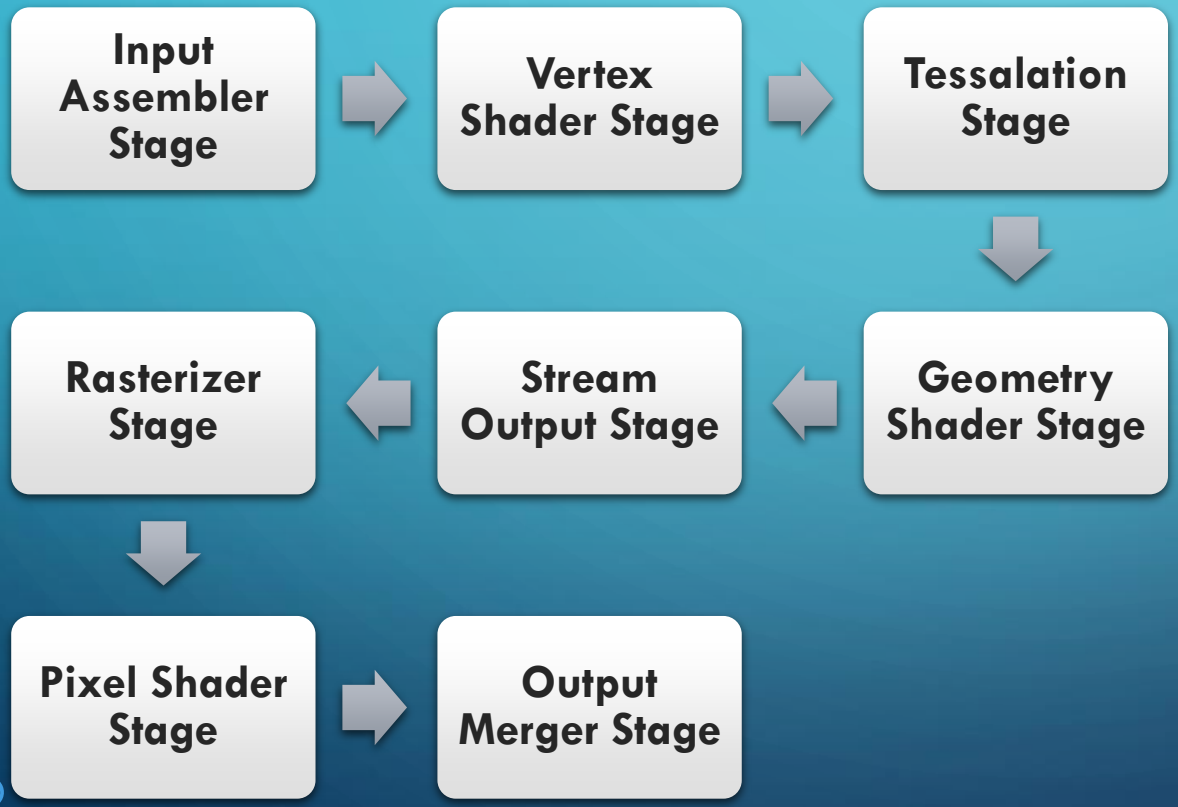


Rasterizer Stage

The rasterizer stage converts vector information (composed of shapes or primitives) into a raster image (composed of pixels) for the purpose of displaying real-time 3D graphics.

DirecX Rendering Pipeline

The Rendering or Graphics Pipeline refers to the sequence of steps used to create 2D representation of a 3D scene.

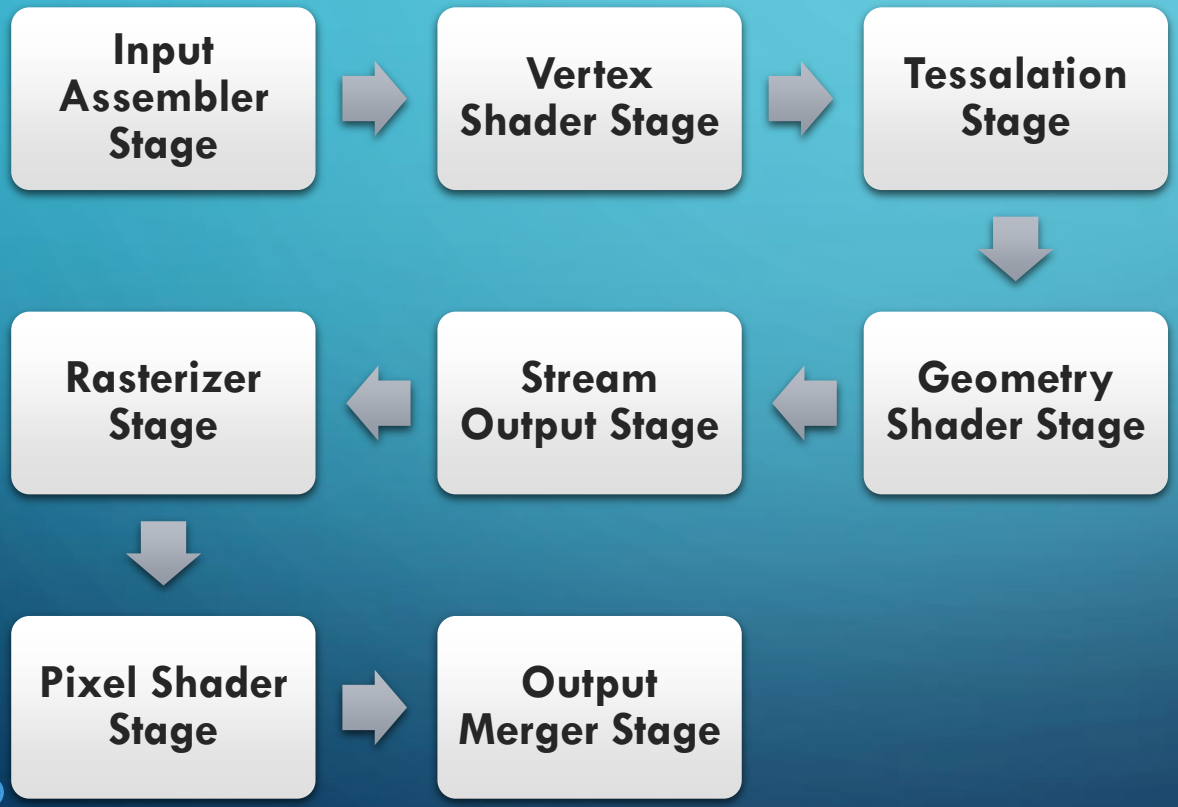


Pixel Shader Stage

A pixel shader is a program that combines constant variables, texture data, interpolated per-vertex values, and other data to produce per-pixel outputs.

DirecX Rendering Pipeline

The Rendering or Graphics Pipeline refers to the sequence of steps used to create 2D representation of a 3D scene.

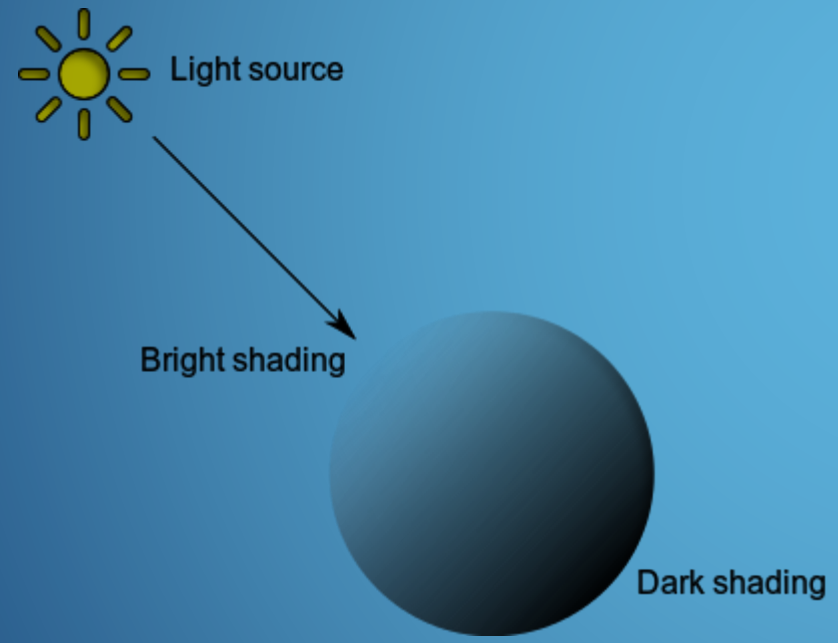


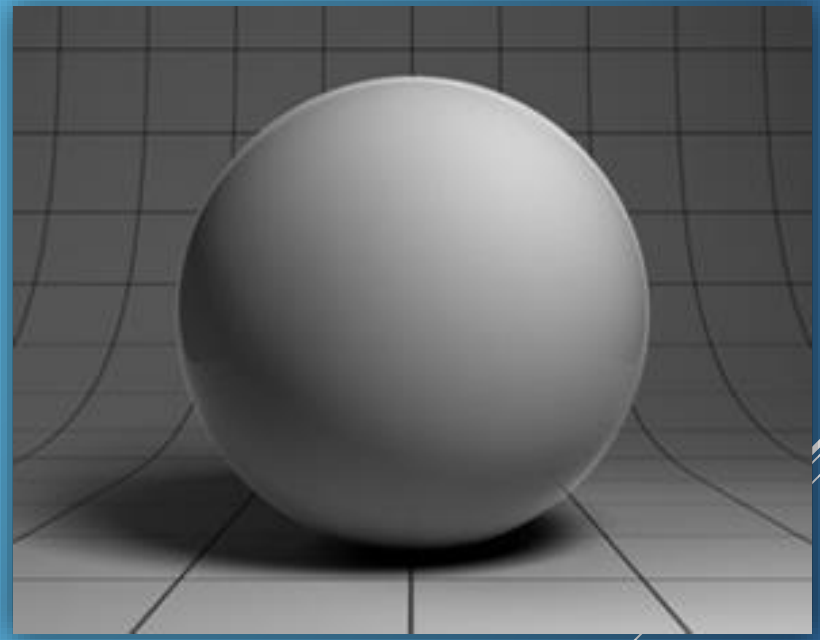
Output Merger Stage

The output-merger (OM) stage generates the final rendered pixel color. The OM stage is the final step for determining which pixels are visible and blending the final pixel colors.

Lighting

In order to calculate the shading of a 3D object, a game engine needs to know the intensity, direction and color of the light that falls on it. These properties are provided by light objects in the scene.



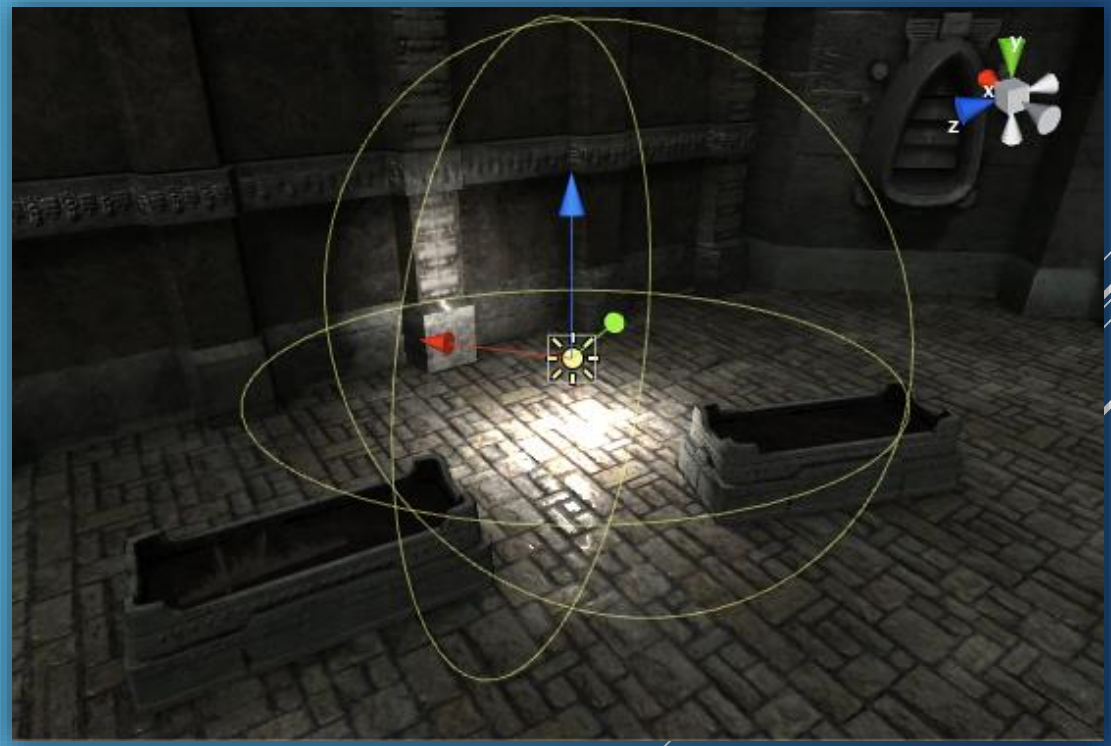


Lighting

Types of Light

Point lights

A point light is located at a point in space and sends light out in all directions equally. The direction of light hitting a surface is the line from the point of contact back to the center of the light object. The intensity diminishes with distance from the light, reaching zero at a specified range. Light intensity is inversely proportional to the square of the distance from the source. This is known as ‘inverse square law’ and is similar to how light behaves in the real world.

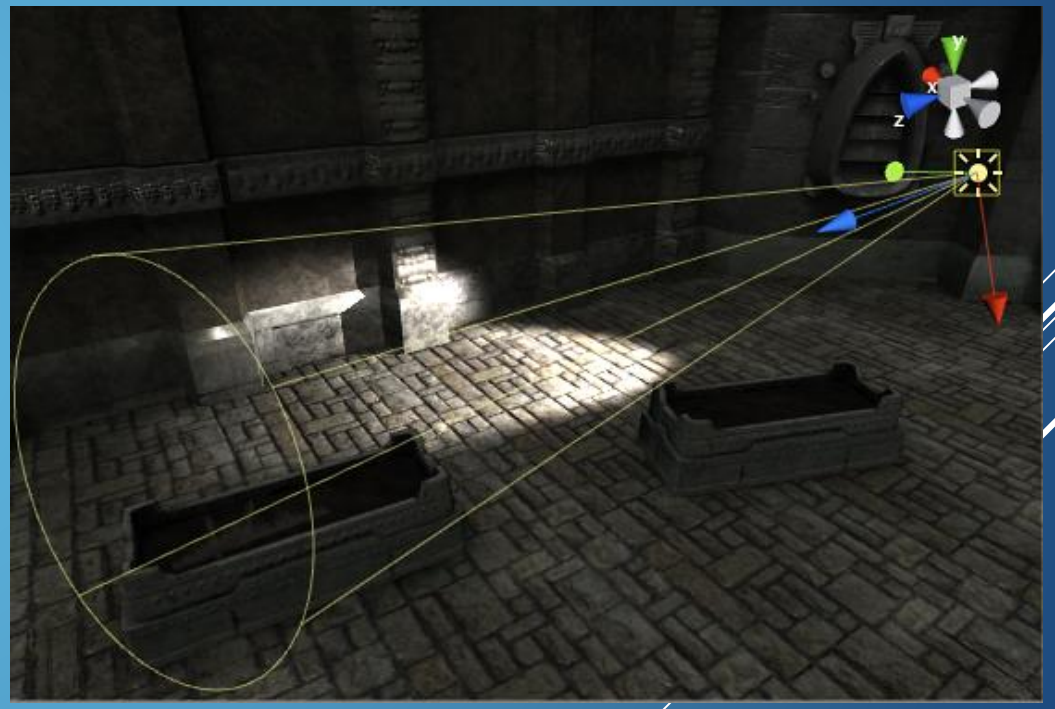


Lighting

Types of Light

Spot lights

Like a point light, a spot light has a specified location and range over which the light falls off. However, the spot light is constrained to an angle, resulting in a cone-shaped region of illumination. The center of the cone points in the forward (Z) direction of the light object. Light also diminishes at the edges of the spot light's cone. Widening the angle increases the width of the cone and with it increases the size of this fade.

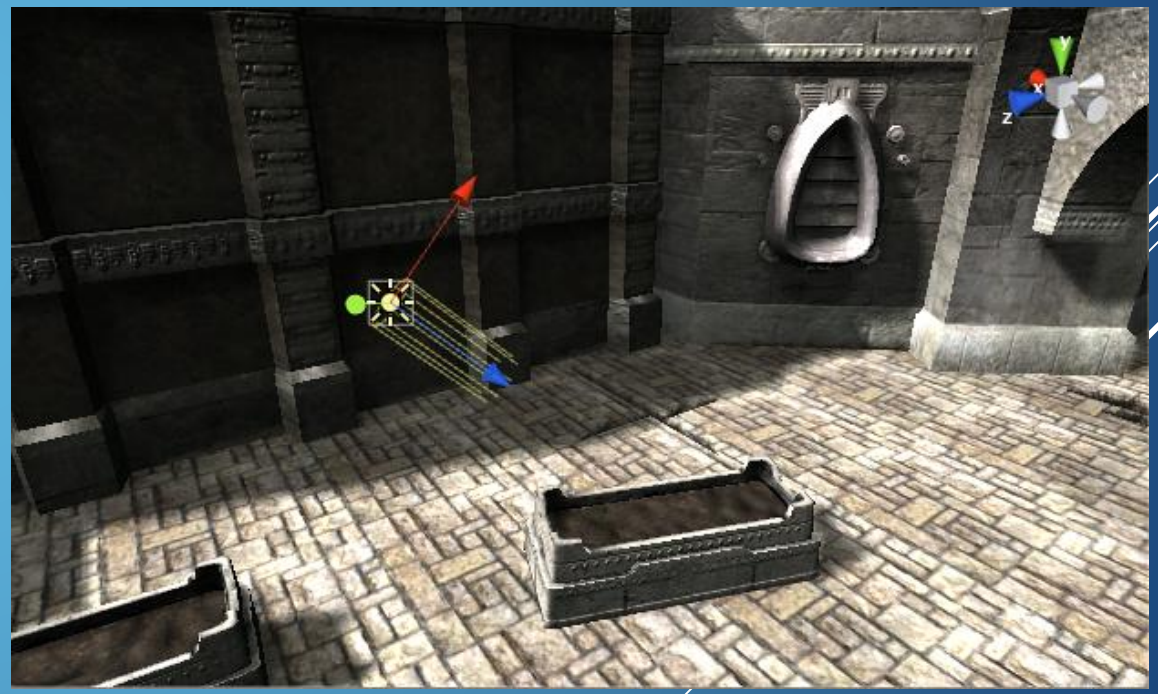


Lighting

Types of Light

Directional lights

Directional lights are very useful for creating effects such as sunlight in your scenes. Behaving in many ways like the sun, directional lights can be thought of as distant light sources which exist infinitely far away. A directional light does not have any identifiable source position and so the light object can be placed anywhere in the scene. All objects in the scene are illuminated as if the light is always from the same direction. The distance of the light from the target object is not defined and so the light does not diminish.



Lighting

Types of Light

Ambient light/Skylight

Ambient light is light that is present all around the scene and doesn't come from any specific source object. It can be an important contributor to the overall look and brightness of a scene.

Lighting

Choosing a Lighting Technique

Realtime lighting

Realtime lighting refers to the lighting technique that contributes direct light to the scene and updates every frame. As lights and objects are moved within the scene, lighting will be updated immediately.

Realtime lighting is the most basic way of lighting objects within the scene and is useful for illuminating characters or other movable geometry.

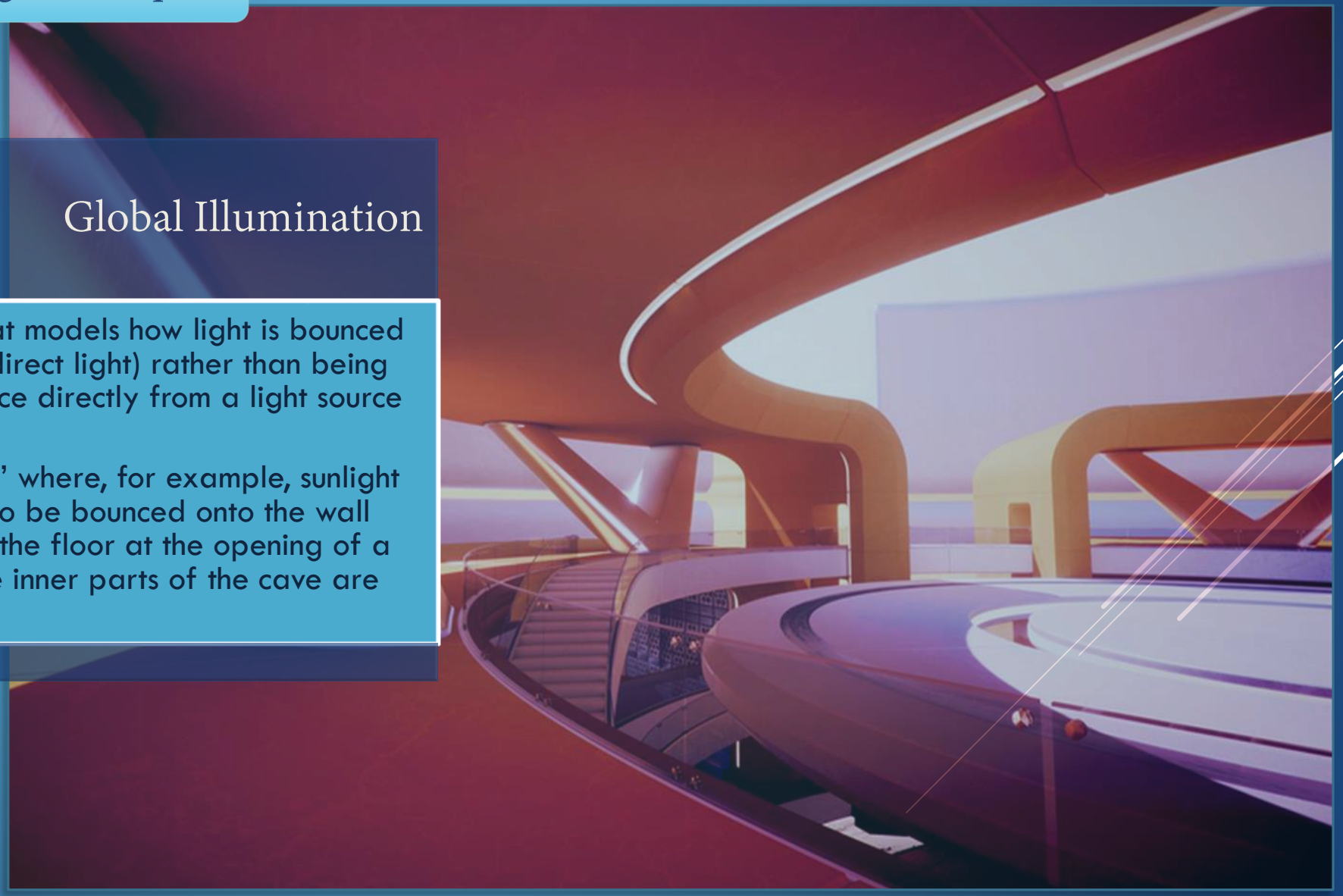
Lighting

Choosing a Lighting Technique

Global Illumination

Global Illumination (GI) is a system that models how light is bounced off of surfaces onto other surfaces (indirect light) rather than being limited to just the light that hits a surface directly from a light source (direct light).

One classic example is ‘color bleeding’ where, for example, sunlight hitting a red sofa will cause red light to be bounced onto the wall behind it. Another is when sunlight hits the floor at the opening of a cave and bounces around inside so the inner parts of the cave are illuminated too.



Lighting

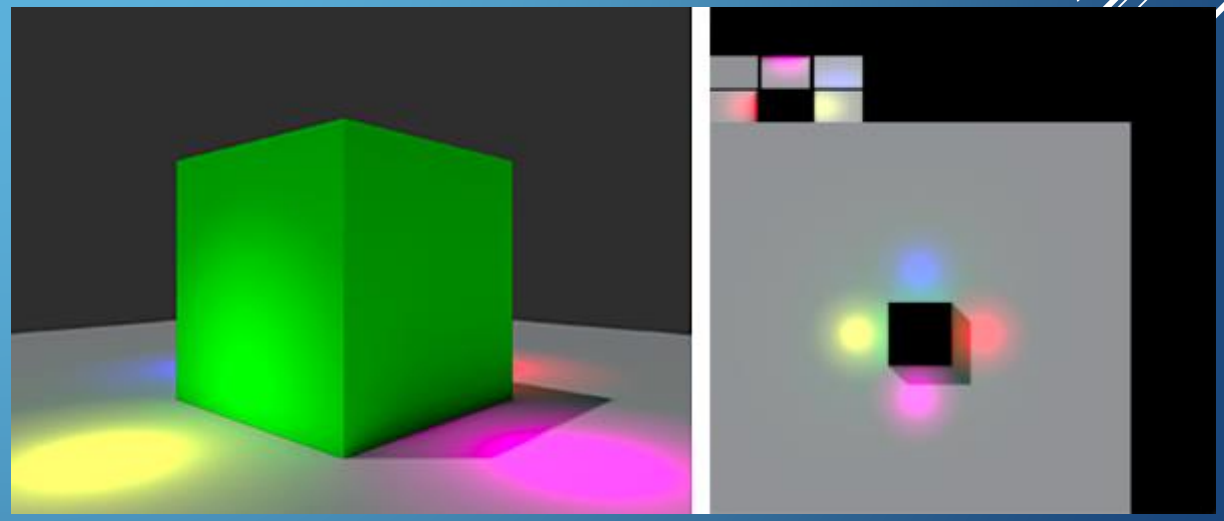
Choosing a Lighting Technique

Baked Lightmaps

Complex static lighting effects can be calculated using global illumination and can be stored in a reference texture map called a lightmap. This calculation process is referred to as baking.

With baked lighting, these lightmaps cannot change during gameplay and so are referred to as ‘static’.

With this approach, we trade the ability to move our lights at gameplay for a potential increase in performance, suiting less powerful hardware such as mobile platforms.



Lighting

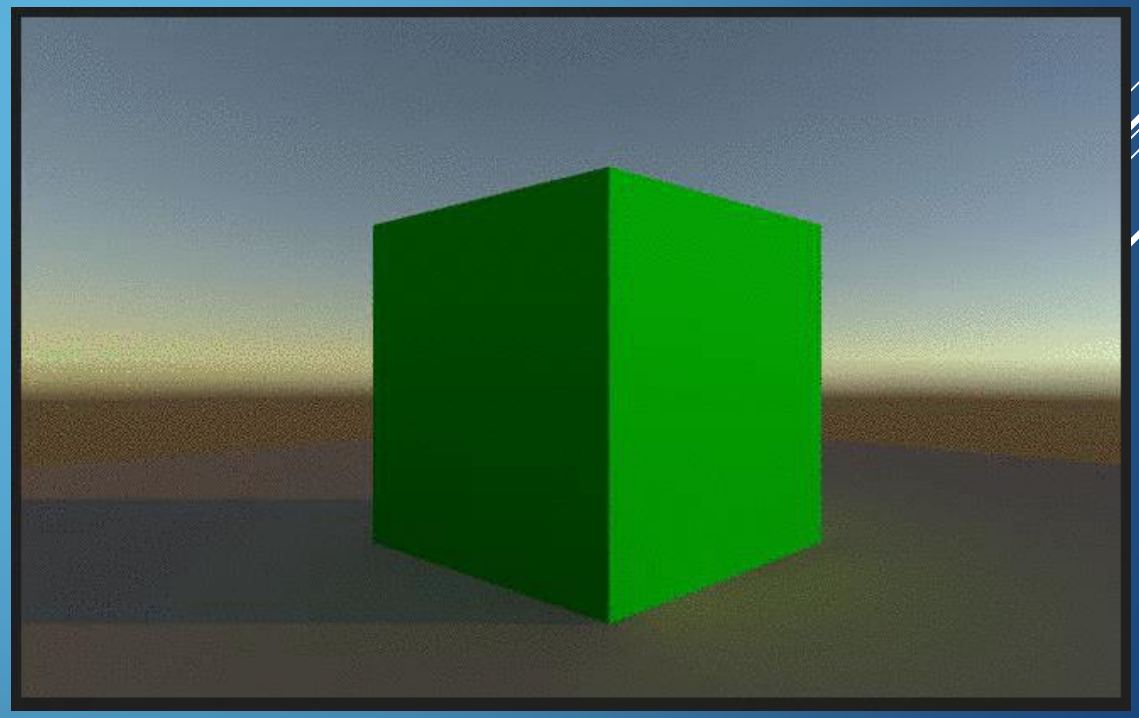
Choosing a Lighting Technique

Precomputed Realtime GI

Whilst static lightmaps are unable to react to changes in lighting conditions within the scene, precomputed realtime GI does offer us a technique for updating complex scene lighting interactively.

With this approach it is possible to create lit environments featuring rich global illumination with bounced light which responds, in realtime, to lighting changes.

A good example of this would be a time of day system - where the position and color of the light source changes over time. With traditional baked lighting, this is not possible.



Lighting

Choosing a Rendering Path

Forward Rendering

In Forward Rendering, each object is rendered in a ‘pass’ for each light that affects it. Therefore each object might be rendered multiple times depending upon how many lights are within range.

The advantages of this approach is that it can be very fast - meaning hardware requirements are lower than alternatives.

However, a significant disadvantage of the forward path is that we have to pay a render cost on a per-light basis. That is to say, the more lights affecting each object, the slower rendering performance will become.

Lighting

Choosing a Rendering Path

Deferred Rendering

In 'Deferred' rendering, we defer the shading and blending of light information until after a first pass over the screen. We then composite these results together with the lighting pass.

This approach has the principle advantage that the render cost of lighting is proportional to the number of pixels that the light illuminates, instead of the number of lights themselves.

Deferred Rendering gives highly predictable performance characteristics, but generally requires more powerful hardware. It is also not supported by certain mobile hardware.

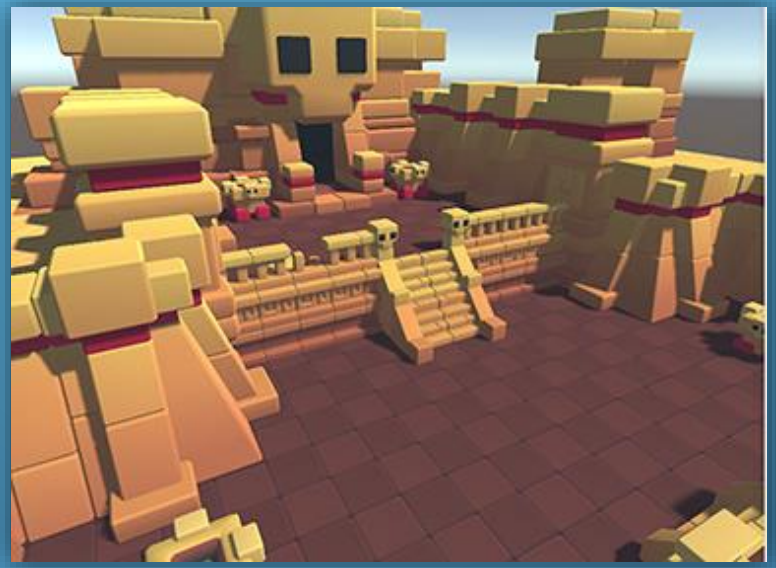
Cameras

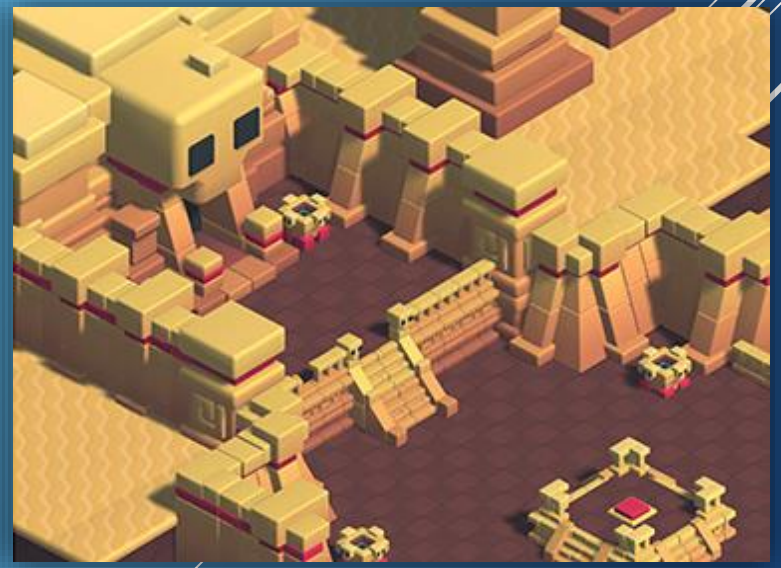
A camera is an object that defines a view in scene space. The object's position defines the viewpoint, while the forward (Z) and upward (Y) axes of the object define the view direction and the top of the screen, respectively.

Perspective and Orthographic Cameras

Perspective - A camera in the real world, or indeed a human eye, sees the world in a way that makes objects look smaller the farther they are from the point of view distance is referred to as *perspective* . This is important for creating a realistic scene.

Orthographic -A camera that does not diminish the size of objects with distance is referred to as *orthographic* .



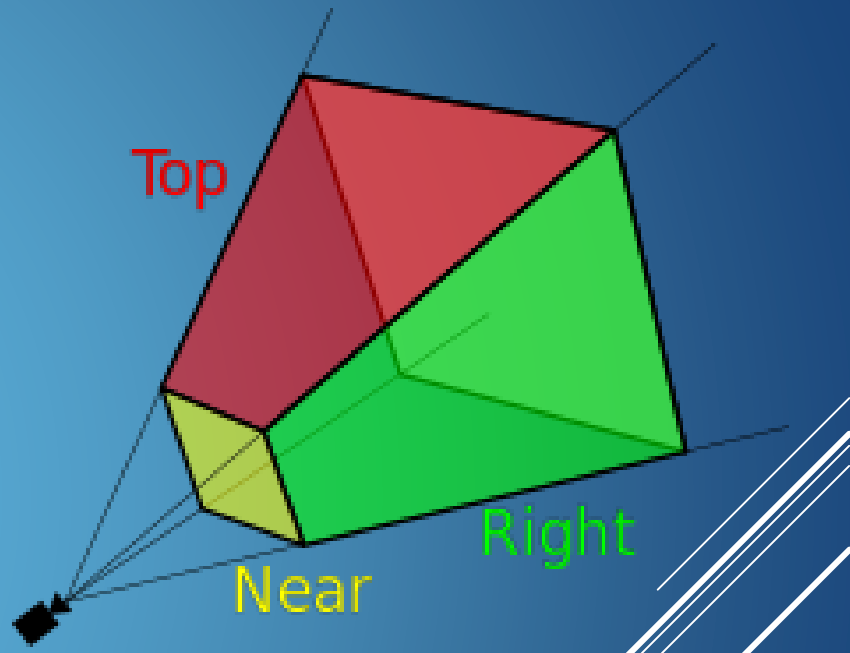


Cameras

Frustum View

The word frustum refers to a solid shape that looks like a pyramid with the top cut off parallel to the base. This is the shape of the region that can be seen and rendered by a perspective camera.

Any point in a camera's image actually corresponds to a line in world space and only a single point along that line is visible in the image. The outer edges of the image are defined by the diverging lines that correspond to the corners of the image. The near and far clipping planes are parallel to the camera's XY plane and each set at a certain distance along its center line. Anything closer to the camera than the near clipping plane and anything farther away than the far clipping plane will not be rendered.



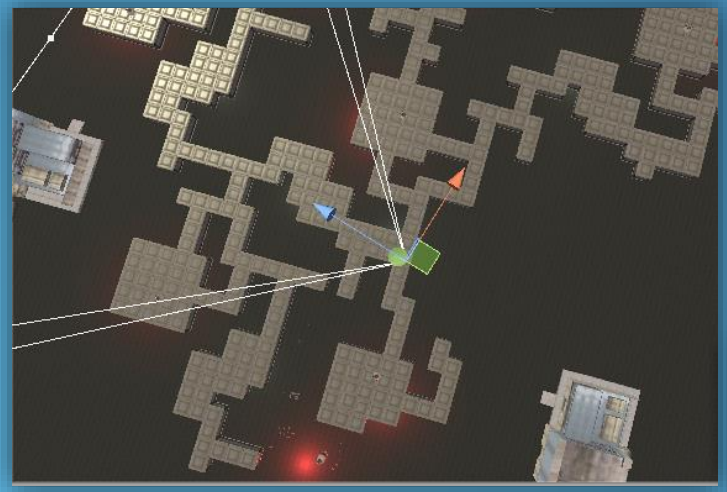
Field of View(FOV) – FOV is the extent of the observable game world that is seen on the display at a given moment.

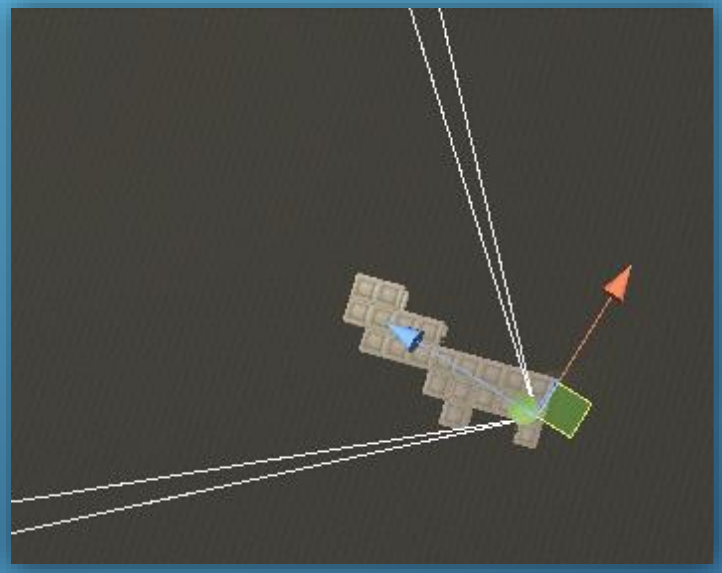
Cameras

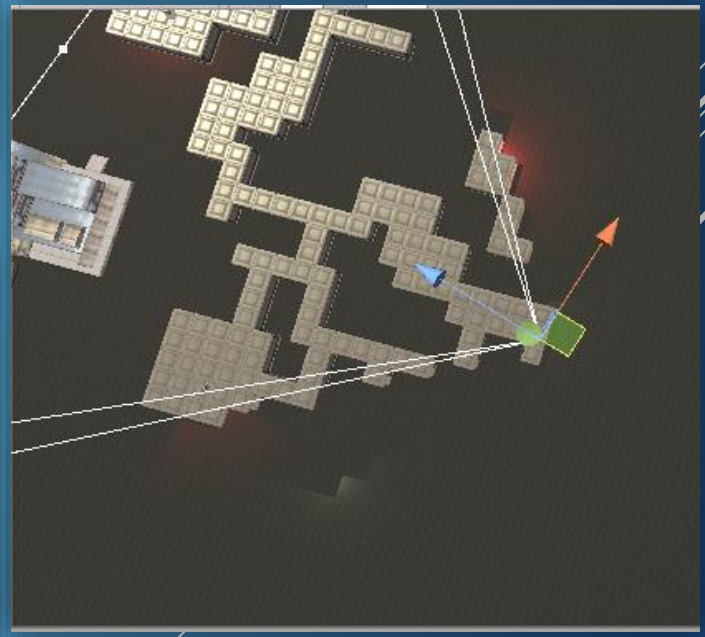
Occlusion Culling

Occlusion Culling is a feature that disables rendering of objects when they are not currently seen by the camera because they are occluded by other objects. This does not happen automatically in 3D computer graphics since most of the time objects farthest away from the camera are drawn first and closer objects are drawn over the top of them (this is called “overdraw”).

The occlusion culling process will go through the scene using a virtual camera to build a hierarchy of potentially visible sets of objects. This data is used at runtime by each camera to identify what is visible and what is not.





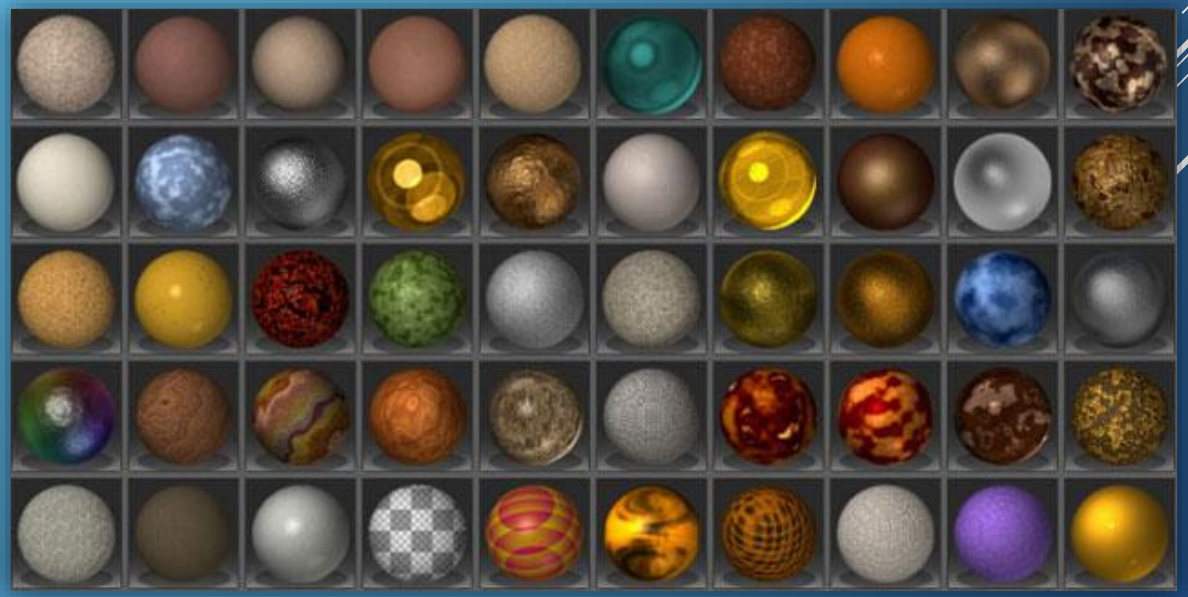


Materials and Shaders

Materials

A Material is an asset that can be applied to a mesh to control the visual look of the scene. A Material literally defines the type of surface from which your object appears to be made. You can define its color, how shiny it is, whether you can see through the object, and much more.

In more technical terms, when light from the scene hits the surface, a Material is used to calculate how that light interacts with that surface. These calculations are done using incoming data that is input to the Material from a variety of images (textures) and math expressions, as well as from various property settings inherent to the Material itself.



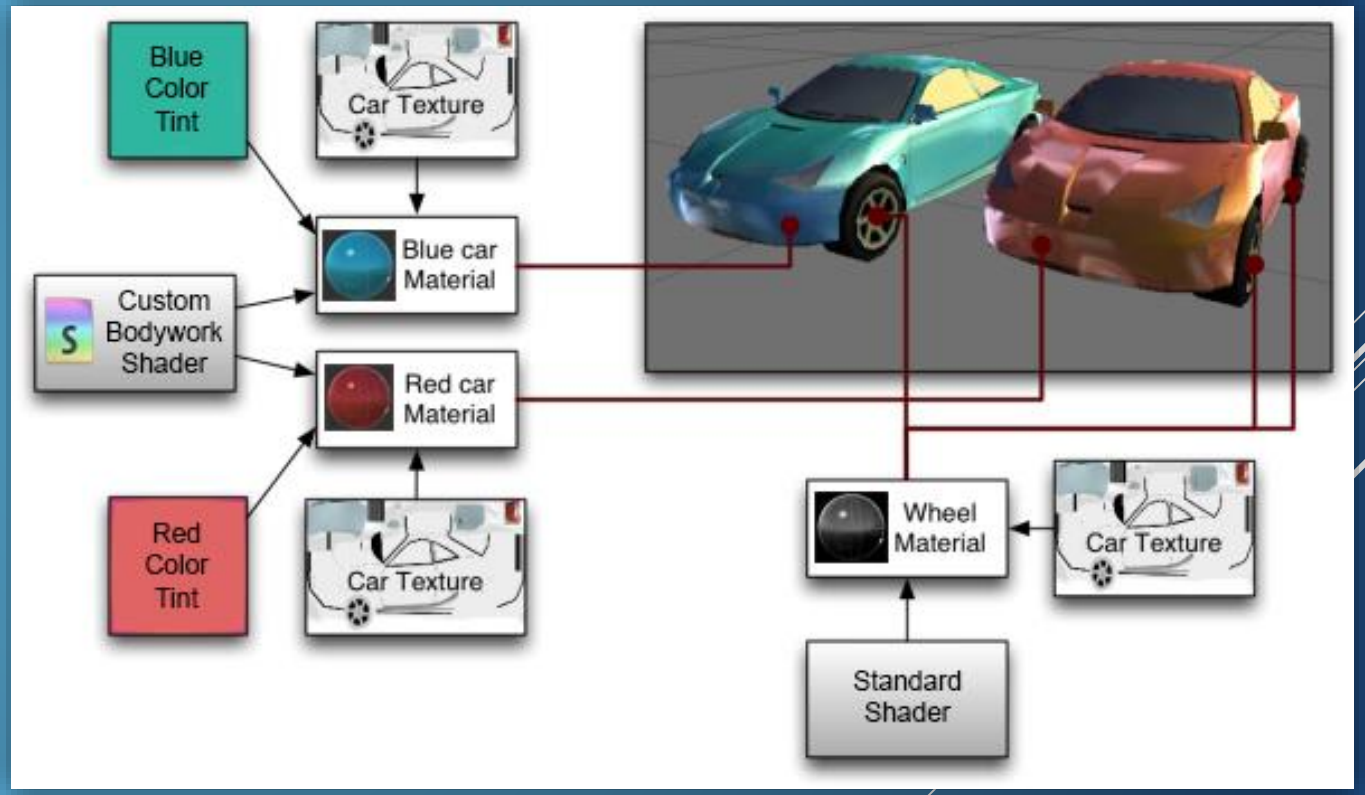
Materials and Shaders

Shaders

A Shader is a script which contains mathematical calculations and algorithms for how the pixels on the surface of a model should look.

Within any given Shader are a number of properties which can be given values by a Material using that shader. These properties can be numbers, colors definitions or textures.

It is possible and often desirable to have several different Materials which may reference the same textures. These materials may also use the same or different shaders, depending on the requirements.



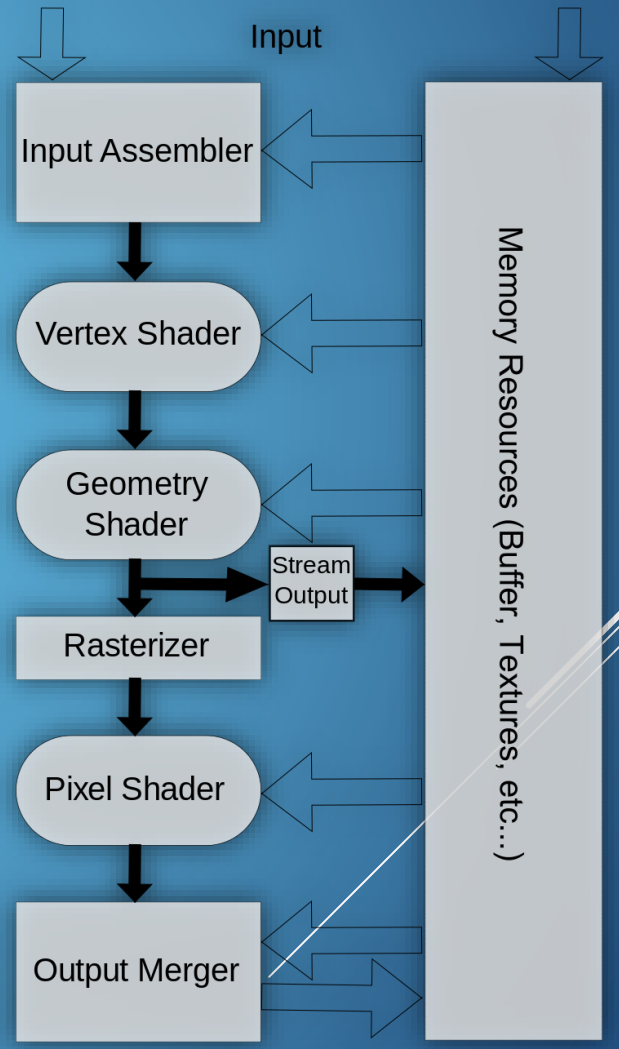
A shader takes the place of two important stages in the Rendering Pipeline: **Vertex Shader** and **Pixel Shader** stage.

Materials and Shaders

HLSL and GLSL Shaders

HLSL is the High Level Shading Language for DirectX. Using HLSL, you can create programmable shaders for your 3D applications.

The OpenGL Shading Language (GLSL) is the principle shading language for OpenGL.



Vertex and Pixel Shaders

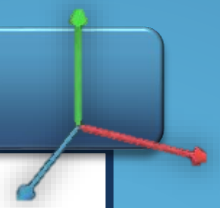
Vertex shaders transform the vertices of a triangle from a local model coordinate system to the screen position.

Pixel(or Fragment) shaders compute the color of a pixel within a triangle rasterized on screen.

Materials and Shaders

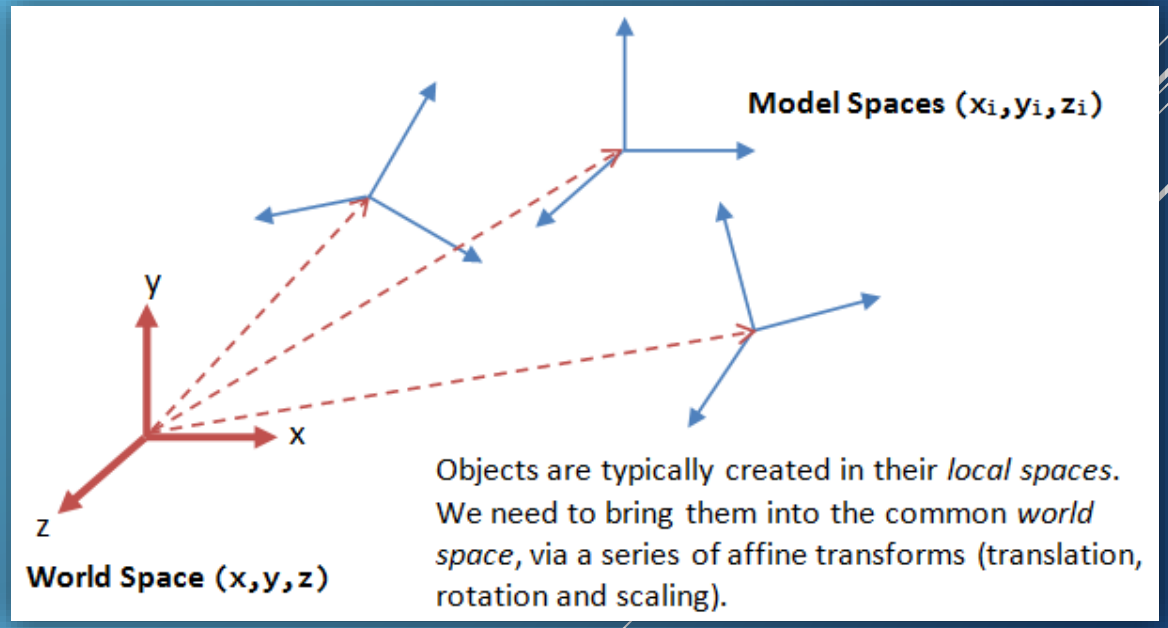
Understanding Transformation Matrices

Vector Spaces: Model Space and World Space



A vector space is a mathematical structure that is defined by a given number of linearly independent vectors. These vectors can be scaled and added together to obtain all the other vectors in the space. 3D models live in one specific vector space, which goes under the name of **Model Space.**

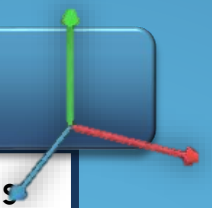
When an artist authors a 3D model he creates all the vertices and faces relatively to the 3D coordinate system of the tool he is working in, which is the Model Space. Every model in the game *lives in its own Model Space* and if you want them to be in any spatial relation (like if you want to put a teapot over a table) you need to transform them into a common space (which is what is often called **World Space**).



Materials and Shaders

Understanding Transformation Matrices

Transformation Matrix



We can see transformations in a vector space simply as a change from one space to another. If we want to represent a transformation from one 3D space to another we will need a **4x4 Matrix**, also called Transformation matrix. In order to apply the transformation we have to multiply all the vectors that we want to transform against the transformation matrix.

X-Rotation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-Rotation in 3D

$$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale in 3D

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(4x4)*(4x1) = (4x1)

Y-Rotation in 3D

$$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Multiplication

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ q \end{bmatrix}$$

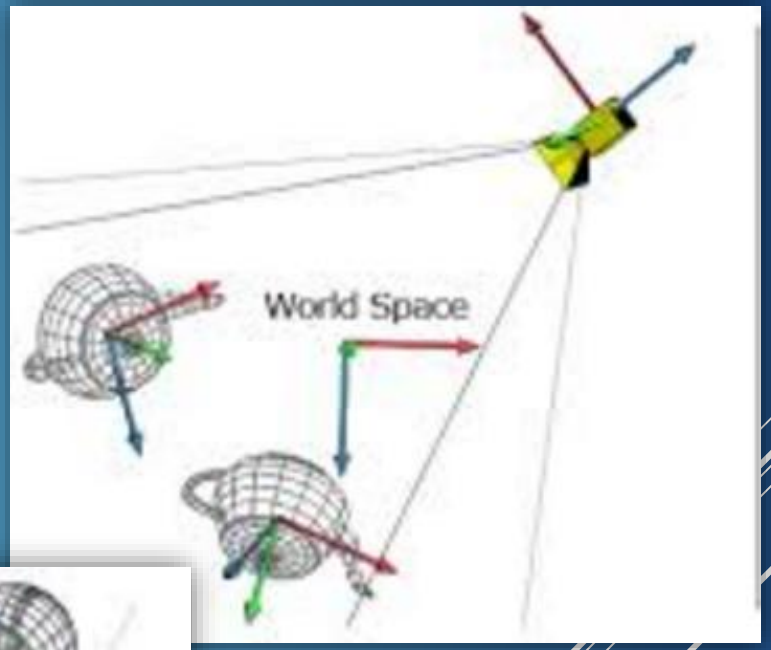
Materials and Shaders

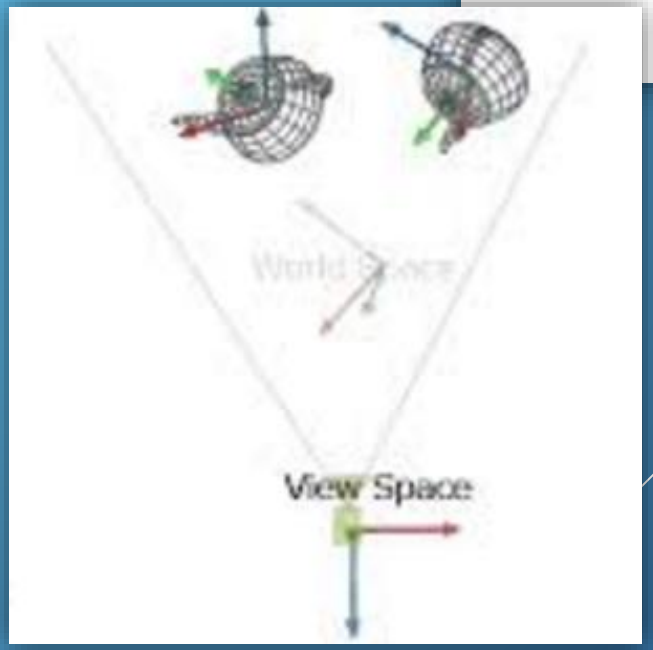
Understanding Transformation Matrices

Model Space, World Space, View Space

The first step when we want to render a 3D scene is to put all the models in the same space, the **World Space**. Since every object will be in its own position and orientation in the world, every one has a different Model to World transformation matrix.

With all the objects at the right place we now need to project them to the screen. This is usually done in two steps. The first step moves all the object in another space called the **View Space**. The View Space is the world as seen by the viewer(or the camera), also called as **Camera Space** or **Eye Space**.





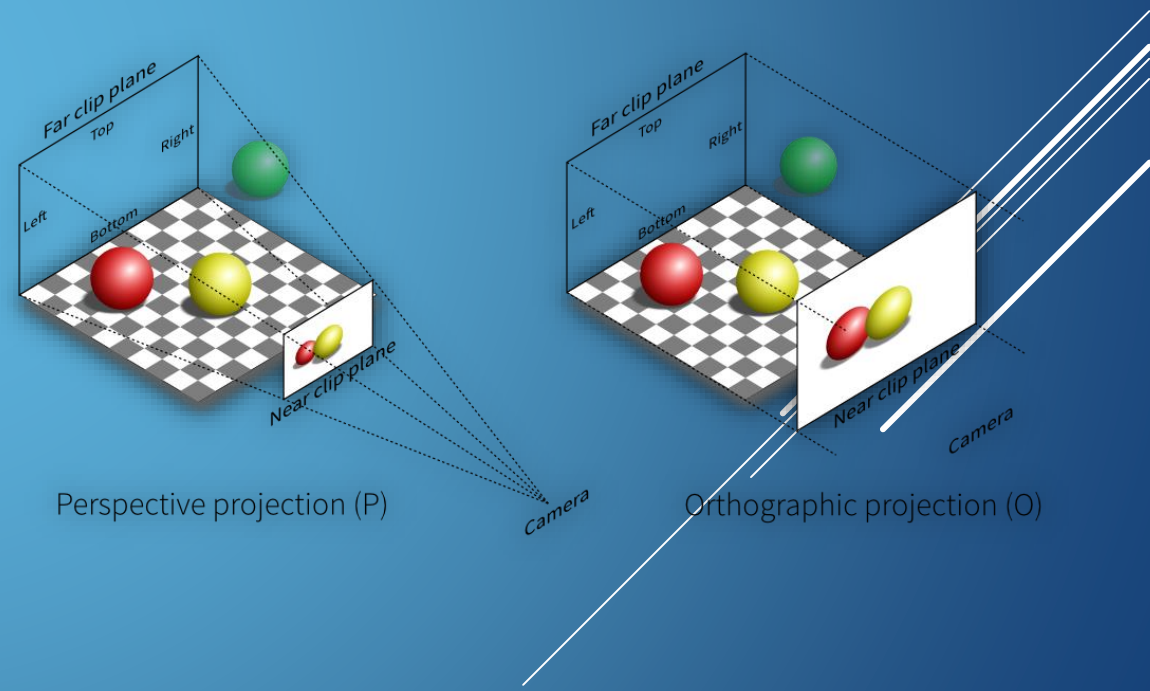
Materials and Shaders

Understanding Transformation Matrices

Projection Space

The projection matrix is typically a scale and perspective projection. The projection transformation converts the viewing frustum into a cuboid shape. Because the near end of the viewing frustum is smaller than the far end, this has the effect of expanding objects that are near to the camera; this is how perspective is applied to the scene.

To go from the View Space into the Projection Space we need another matrix, the View to Projection matrix, and the values of this matrix depend on what type of projection we want to perform. The two most used projections are the **Orthographic Projection** and the **Perspective Projection.**



Materials and Shaders

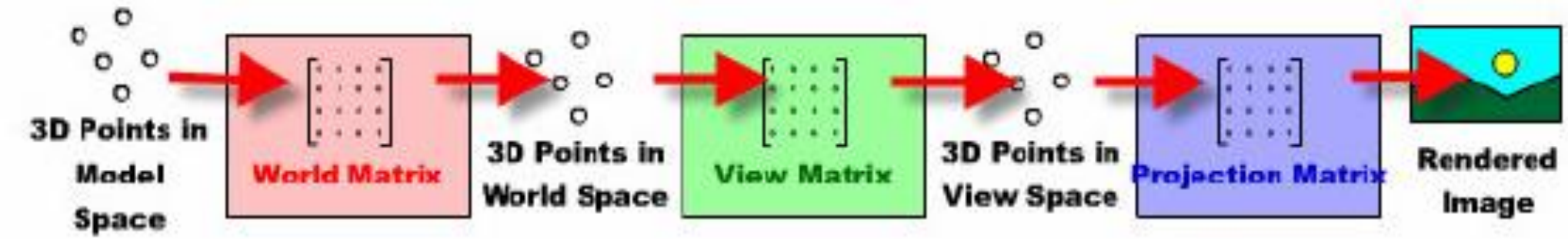
Understanding Transformation Matrices

Model View Projection Matrix

To work with multiple objects, and position each object in the 3D world, we compute a transformation matrix that will:

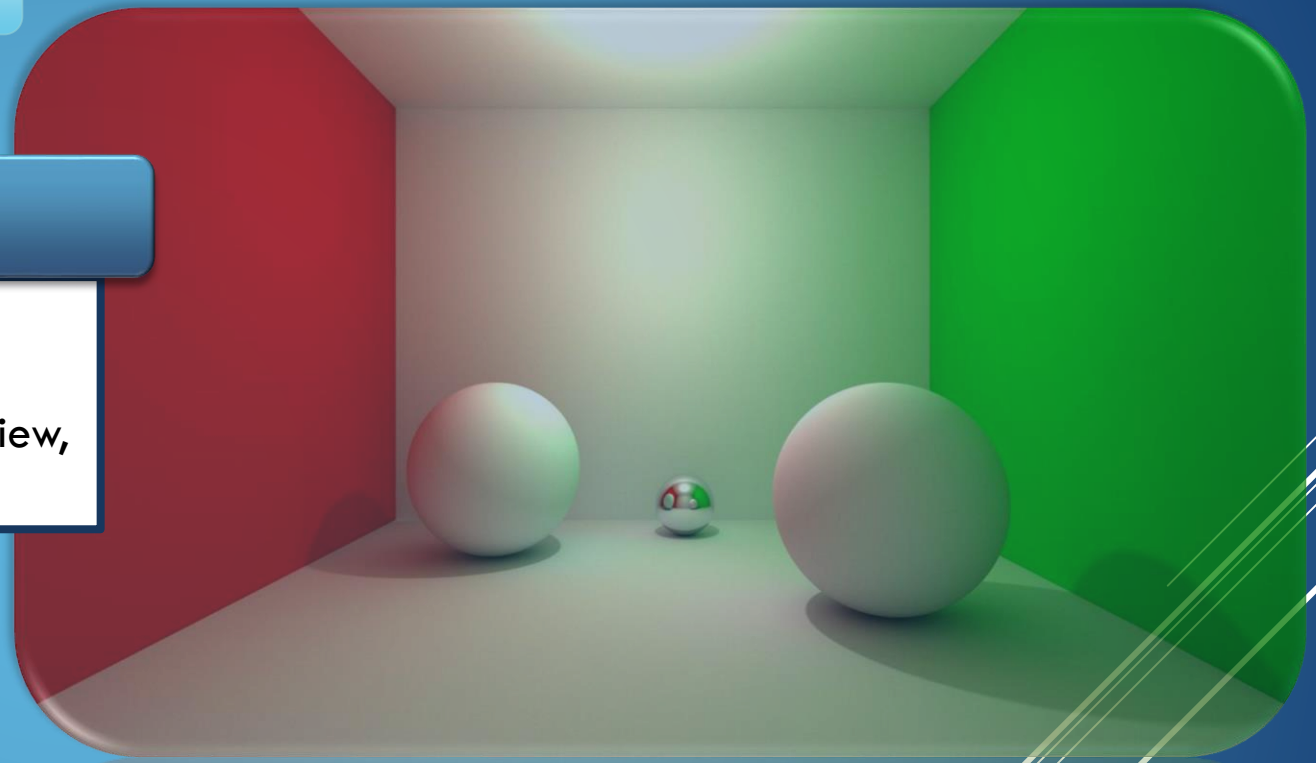
1. shift from model (object) coordinates to world coordinates (model->world)
2. then from world coordinates to view (camera) coordinates (world->view)
3. then from view coordinates to projection (2D screen) coordinates (view->projection)

The goal is to compute a global transformation matrix, called MVP, that we'll apply on each vertex to get the final 2D point on the screen.



Materials and Shaders

Shader Example : Ambient Lighting (HLSL)



Adding Variables

We create 5 variables. The first three are of type *float4x4*. This is basically a 4-by-4 matrix. These three variables will directly correspond to the world, view, and projection matrices in your game.

```
float4x4 World;
float4x4 View;
float4x4 Projection;
float4 AmbientColor = float4(1, 1, 1, 1);
float AmbientIntensity = 0.1;
```

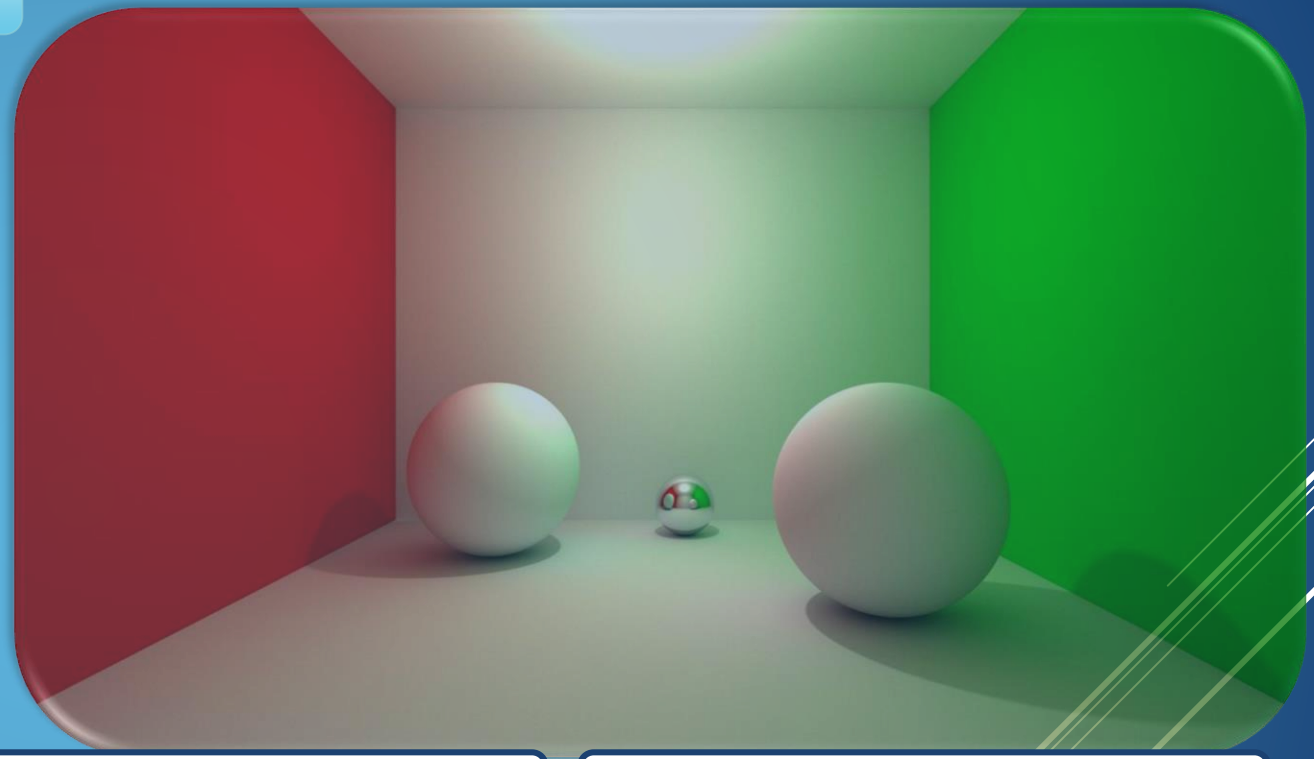
Materials and Shaders

Shader Example : Ambient Lighting (HLSL)

Defining the Data Structures

The next step is for us to create the necessary data structures to store data between three places: between the application and the vertex shader (this is the input to the vertex shader), between the vertex shader and the pixel shader (this is the output of the vertex shader, and also the input to the pixel shader), and the final result, also the output to the pixel shader.

We will first create the structure for the input to the vertex shader, and then the structure for the output from the vertex shader, which is also the input to the pixel shader.



```
struct VertexShaderInput
{
float4 Position : POSITION0;
}
```

```
struct VertexShaderOutput
{
float4 Position : POSITION0;
}
```

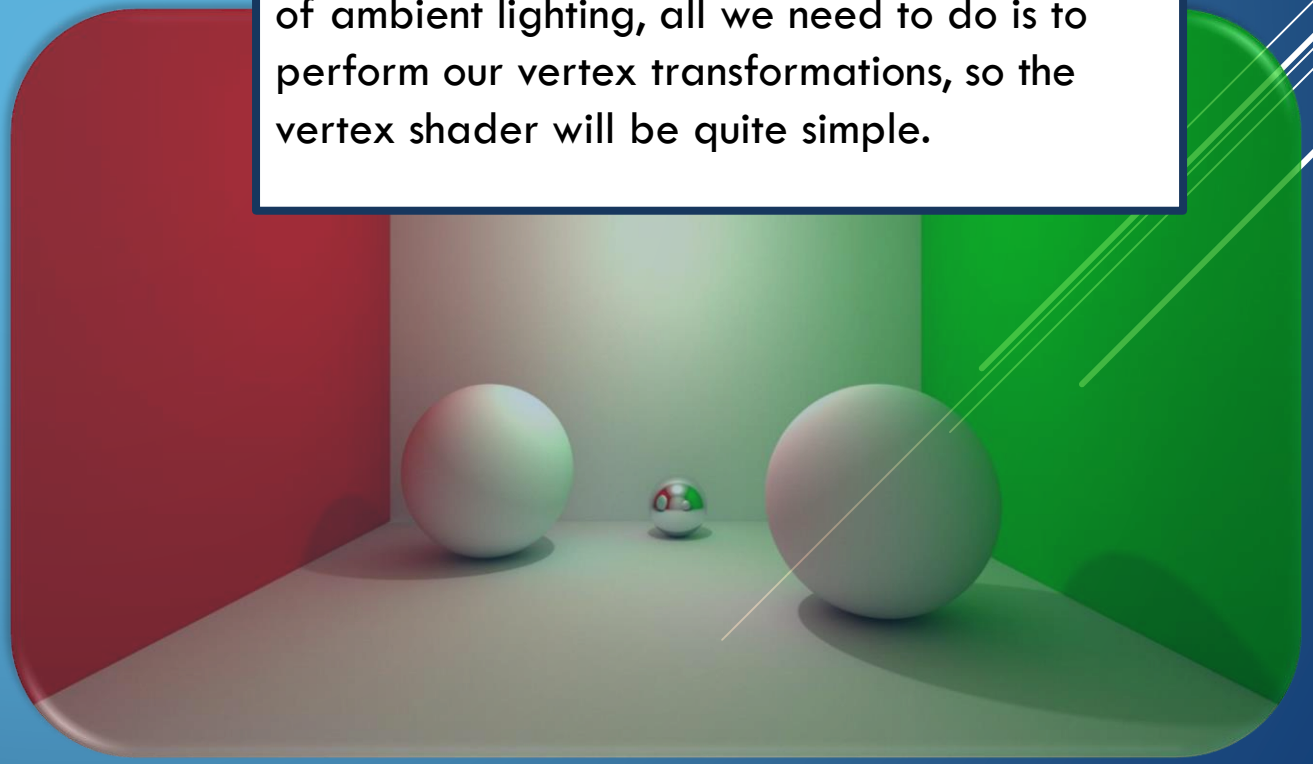
Materials and Shaders

Shader Example : Ambient Lighting (HLSL)

```
VertexShaderOutput VertexShaderFunction(VertexShaderInput input)
{
    VertexShaderOutput output;
    float4 worldPosition = mul(input.Position, World);
    float4 viewPosition = mul(worldPosition, View);
    output.Position = mul(viewPosition, Projection);
    return output;
}
```

Vertex Shader

The next step is to write the vertex shader itself. Remember, the vertex shader has the job of transforming vertices, and doing the necessary lighting calculations so that it can be colored later in the pixel shader. For the case of ambient lighting, all we need to do is to perform our vertex transformations, so the vertex shader will be quite simple.

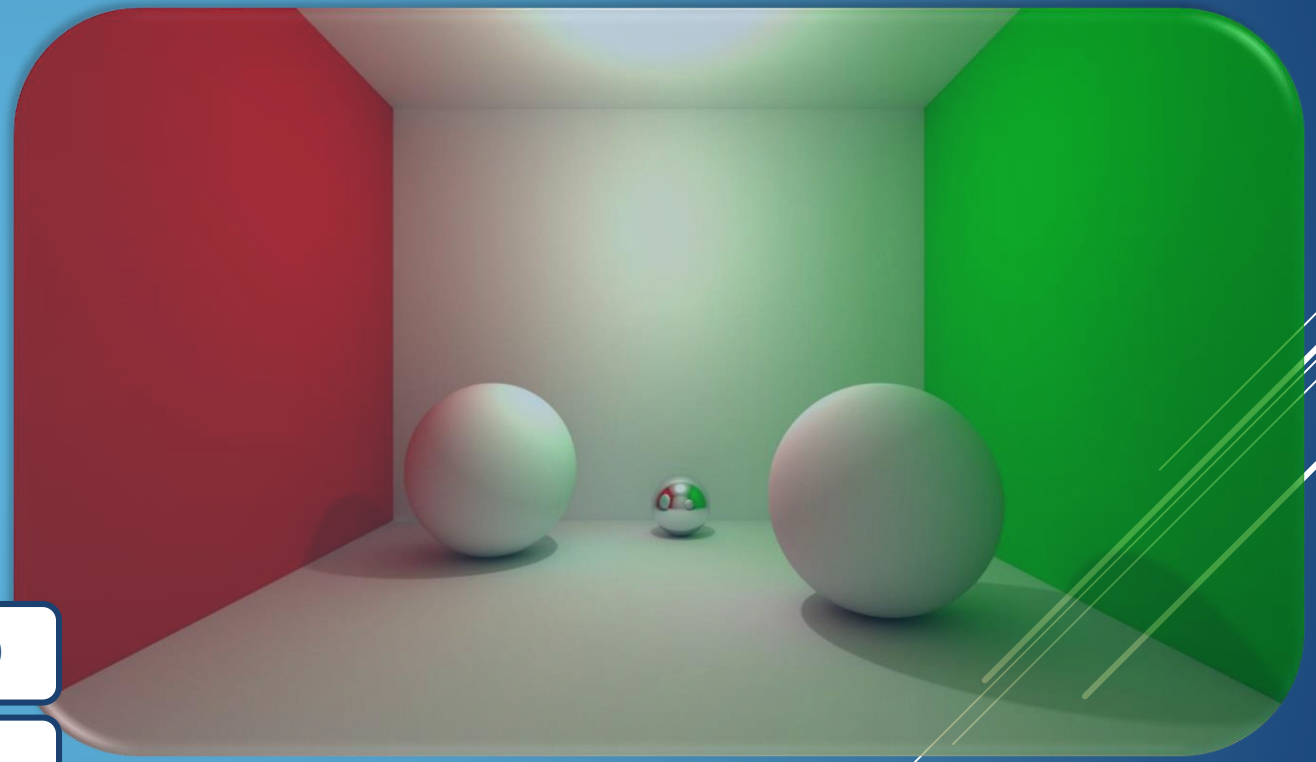


Materials and Shaders

Shader Example : Ambient Lighting (HLSL)

Pixel Shader

This is similar to the vertex shader. Once again, we specify the return type of the function, which in our case is going to be a color, represented by a *float4* (an array of floats).



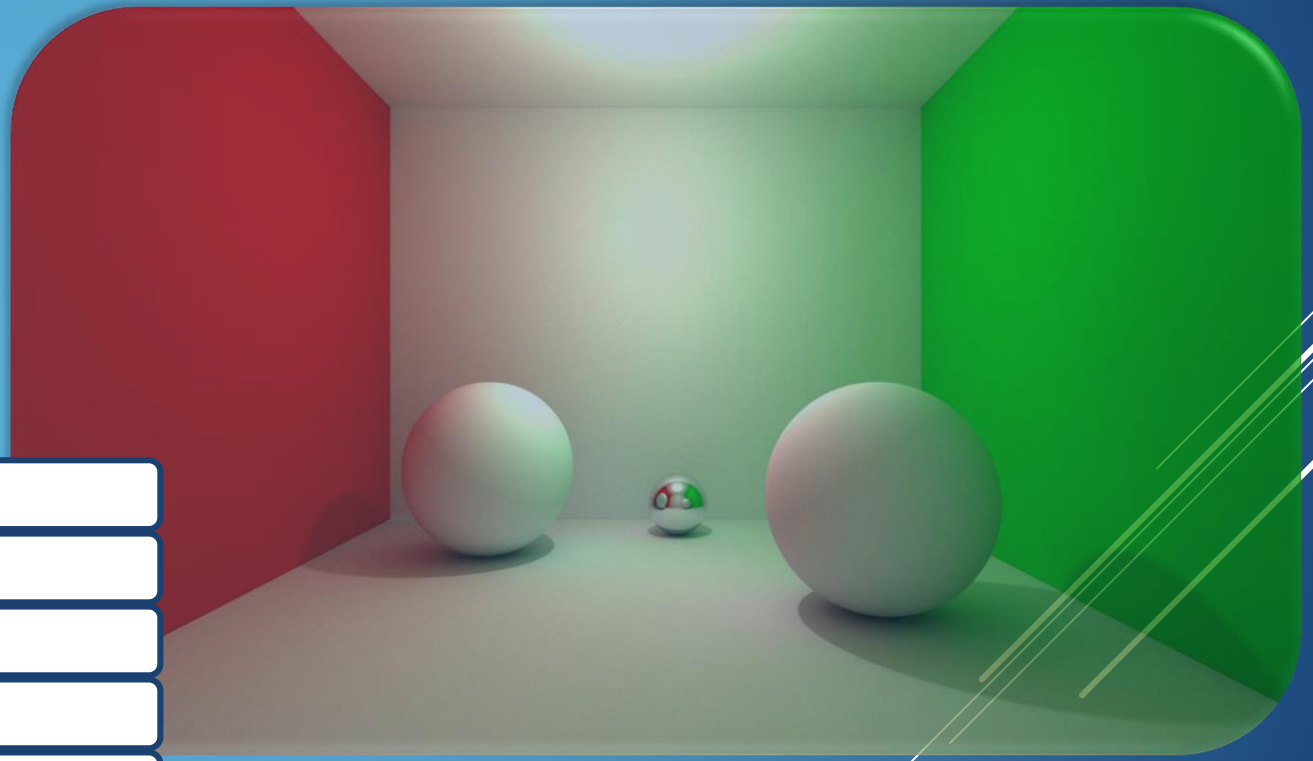
```
float4 PixelShaderFunction(VertexShaderOutput input) : COLOR0
{
    return AmbientColor * AmbientIntensity;
}
```

Materials and Shaders

Shader Example : Ambient Lighting (HLSL)

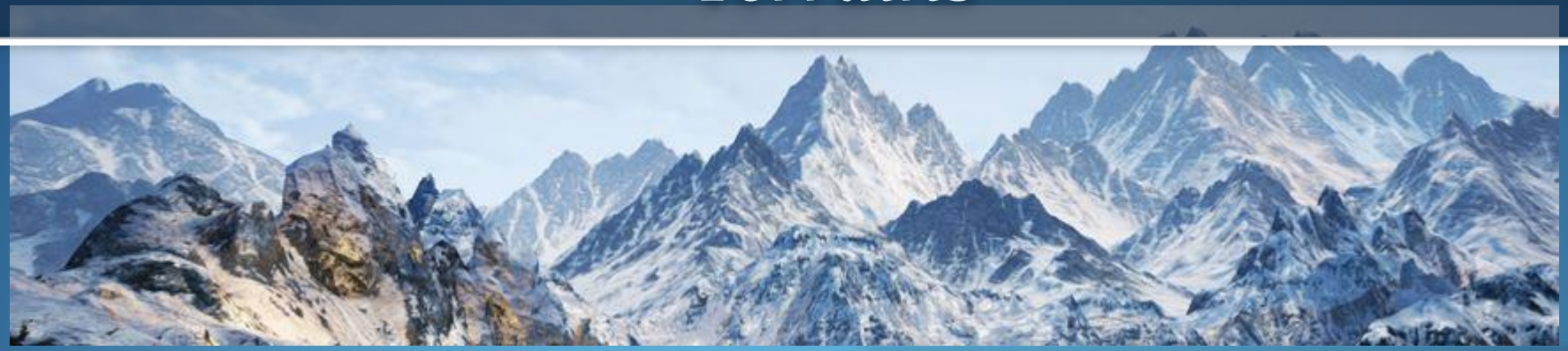
Defining the Technique

The last thing we need to do is define a technique.



```
technique Ambient
{
    pass Pass1
    {
        VertexShader = compile vs_2_0 VertexShaderFunction();
        PixelShader = compile ps_2_0 PixelShaderFunction();
    }
}
```

Terrains



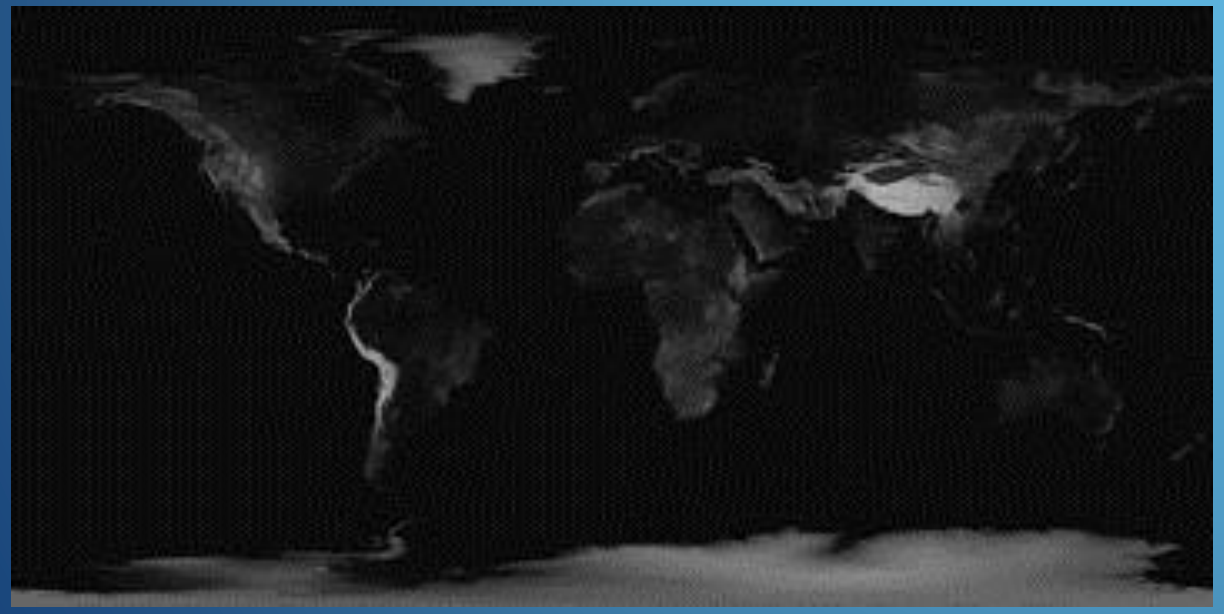
Terrain system allows you to add vast landscapes to your games. You can create terrains from within your game engine or you can auto generate terrains using Heightmaps.

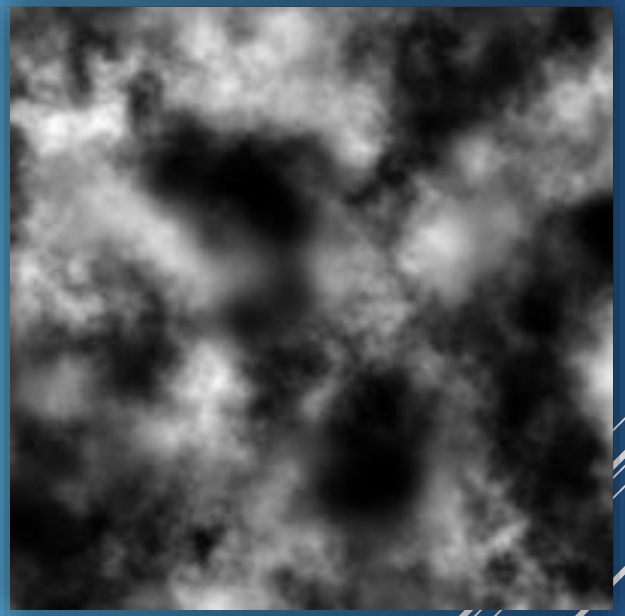
Terrains

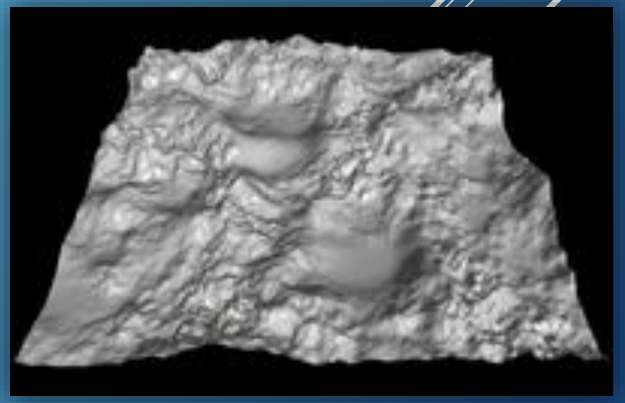
HeightMaps

A Heightmap is a texture map that is used to store surface elevation data ,which can be used to auto generate 3D terrains within a game engine.

A Heightmap is a grayscale image that contains only one channel that is interpreted as the displacement or height from the floor, with black representing minimum height and white representing maximum height





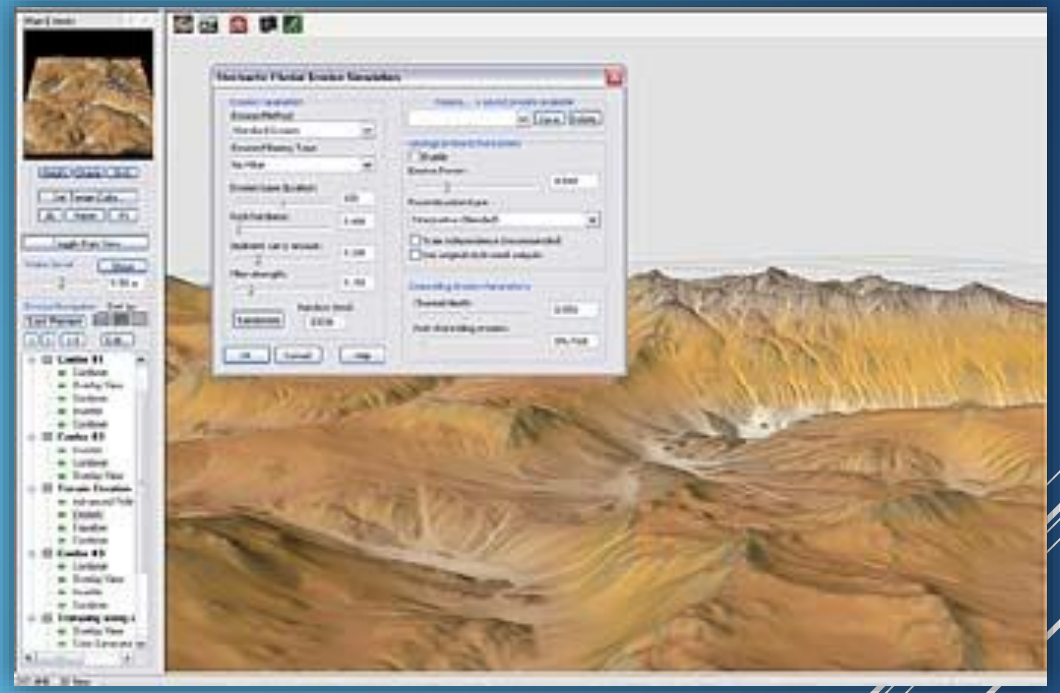


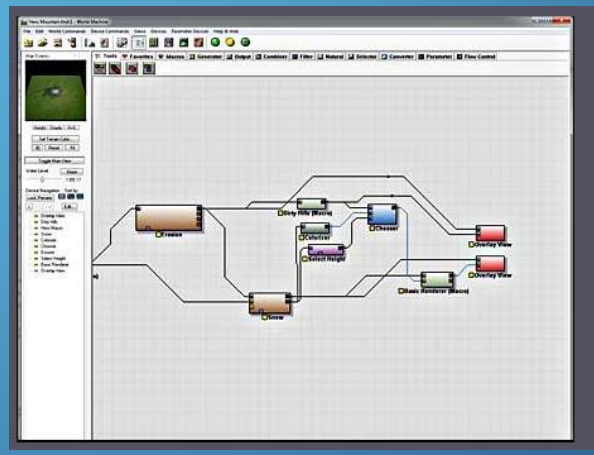
Terrains

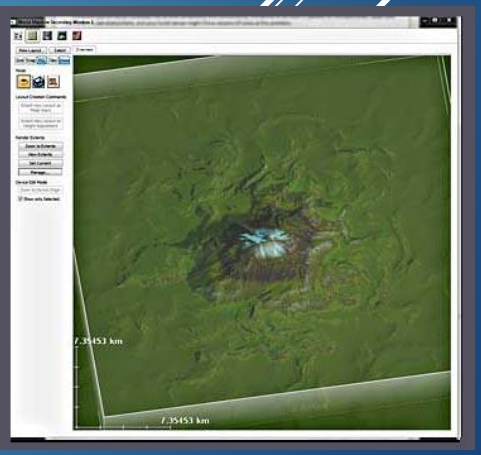
HeightMaps

Creating HeightMaps

Heightmaps can be created by hand with a classical paint program or a special terrain editor like World Machine, which visualizes the terrain in 3D and allows the user to modify the surface. There are tools to raise, lower, smooth or erode the terrain. Another way to create a terrain is to use a terrain generation algorithm. Another method is to use real world data for example from radar satellites or laser/radar measurements from airplanes.







Terrains

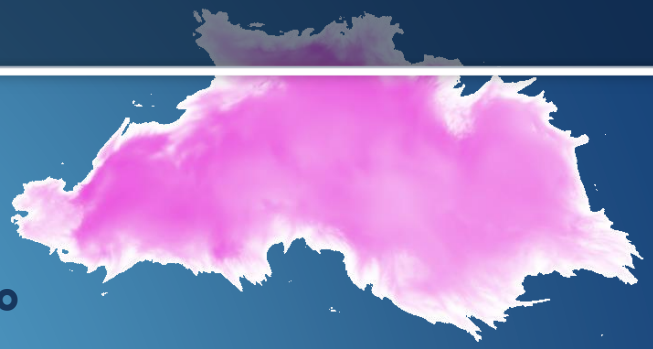
Foliage

Terrains can be furnished with trees and other vegetation. Patches of static meshes can be painted onto a terrain in much the same way that heightmaps and textures are painted but these meshes are solid 3D objects that grow from the surface.

Billboards

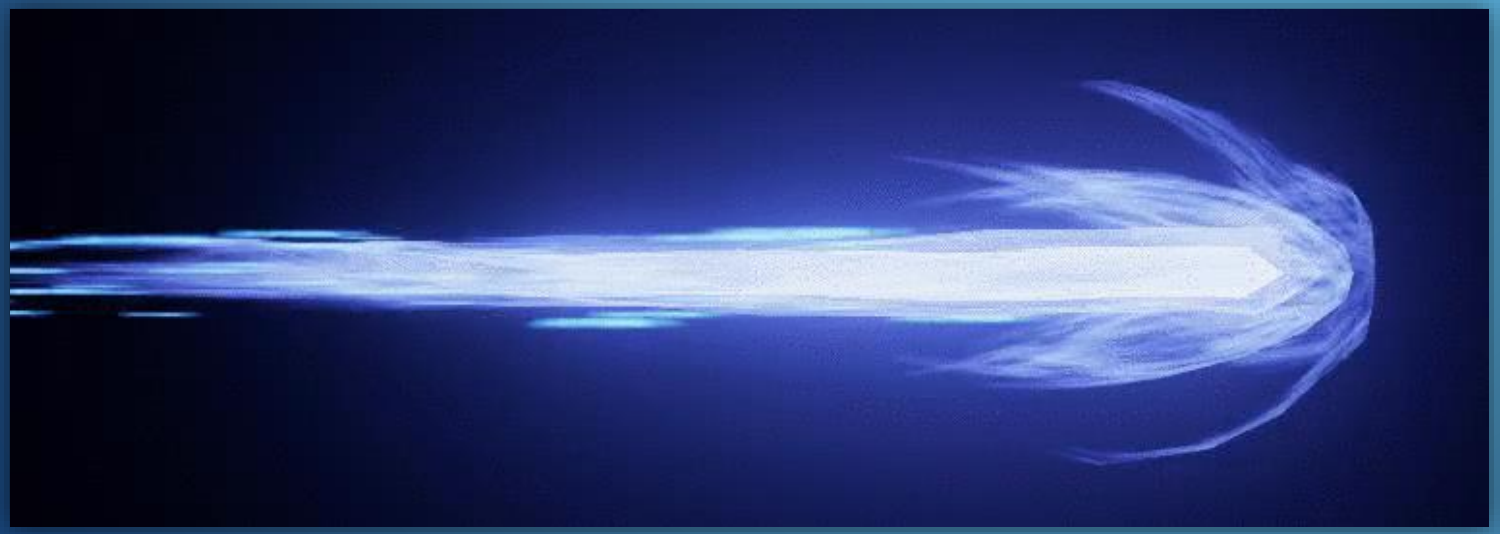
Billboards are 2D images that can be used to along with your 3D foliage meshes to give an illusion of density from a distance without affecting performance.

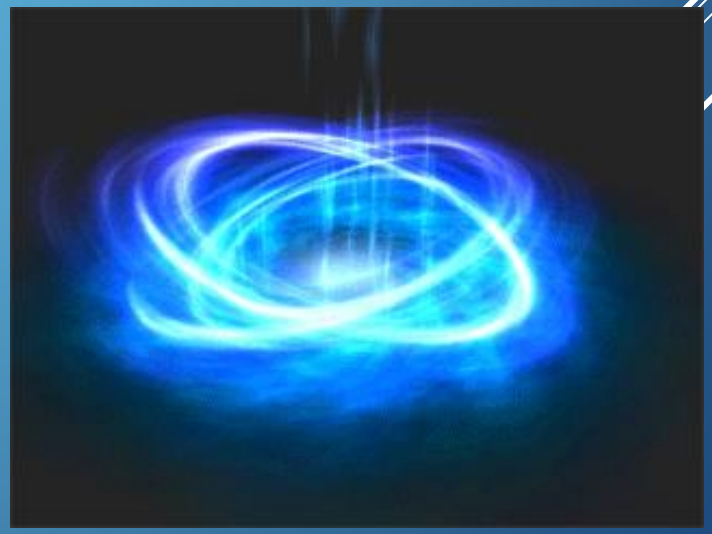
Particle System



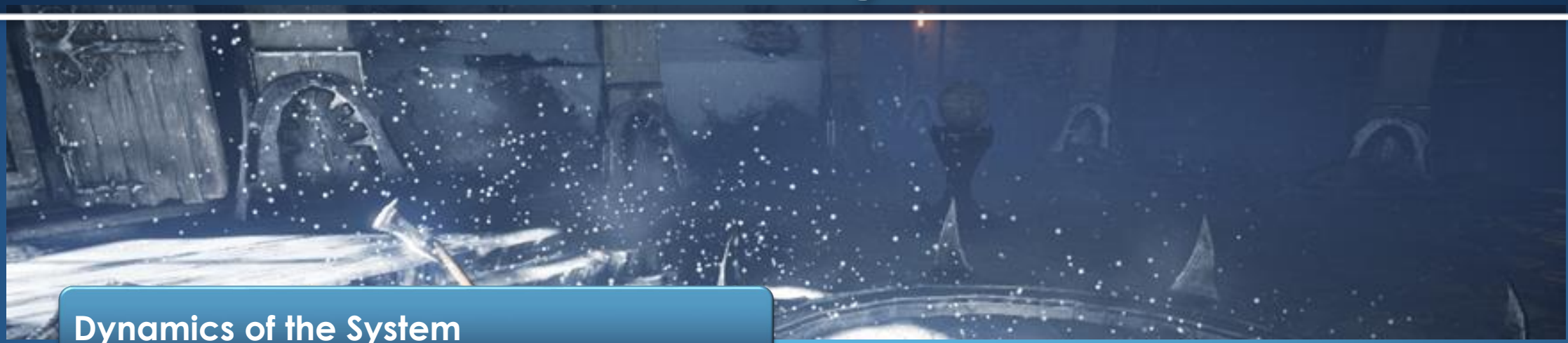
In a 3D game, most characters, props and scenery elements are represented as meshes, while a 2D game uses sprites for these purposes. Meshes and sprites are the ideal way to depict "solid" objects with a well-defined shape. There are other entities in games, however, that are fluid and intangible in nature and consequently difficult to portray using meshes or sprites. For effects like moving liquids, smoke, clouds, flames and magic spells, a different approach to graphics known as particle systems can be used to capture the inherent fluidity and energy.

Particles are small, simple images or meshes that are displayed and moved in great numbers by a particle system.



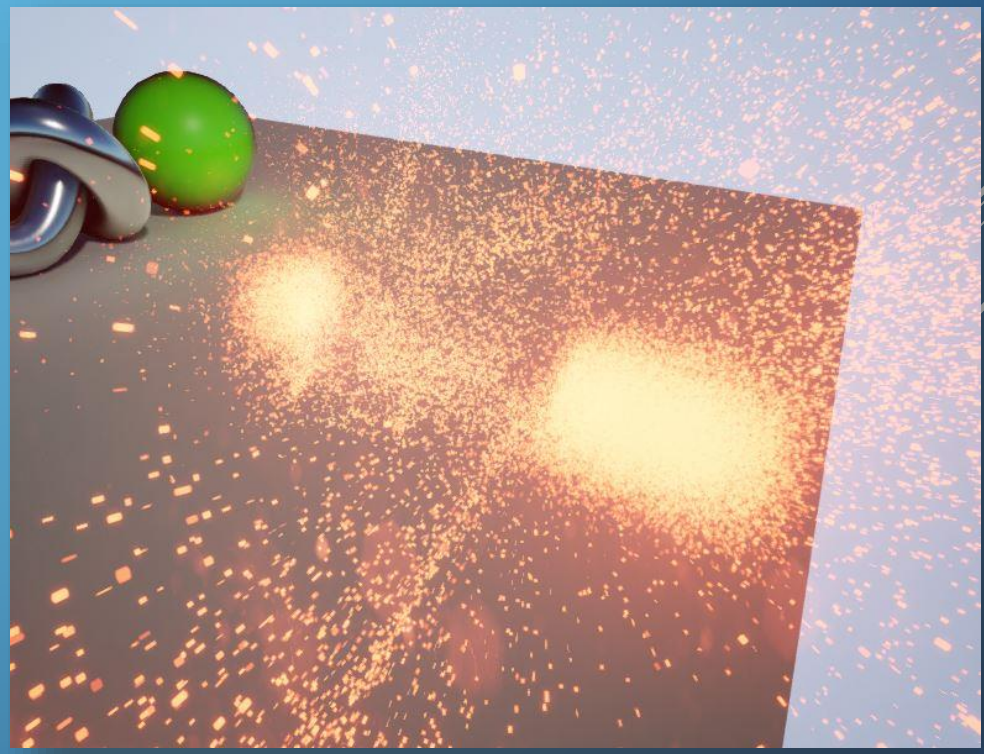


Particle System

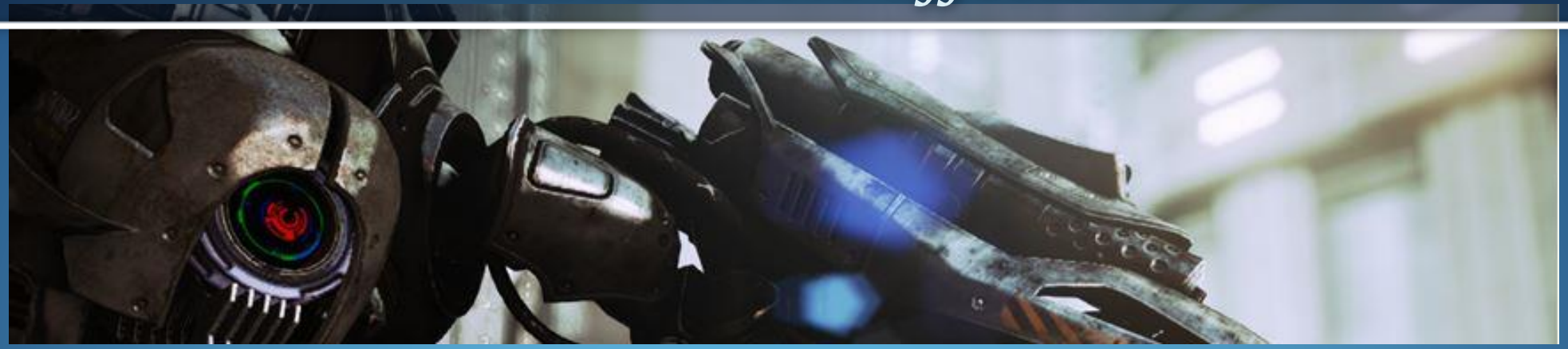


Dynamics of the System

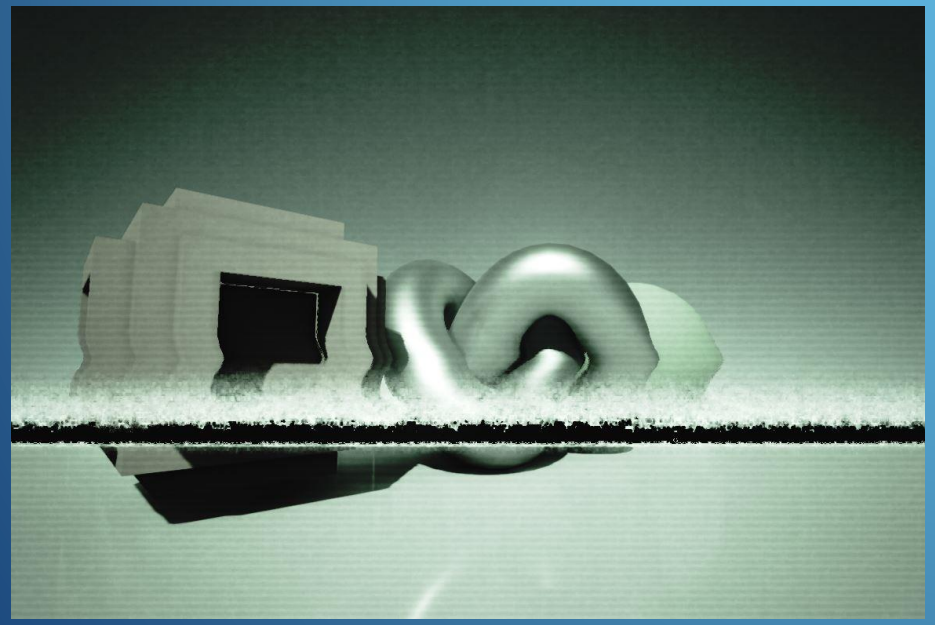
Each particle has a predetermined *lifetime*, typically of a few seconds, during which it can undergo various changes. It begins its life when it is generated or *emitted* by its particle system. The system emits particles at random positions within a region of space shaped like a sphere, hemisphere, cone, box or any arbitrary mesh. The particle is displayed until its time is up, at which point it is removed from the system. The system's *emission rate* indicates roughly how many particles are emitted per second.



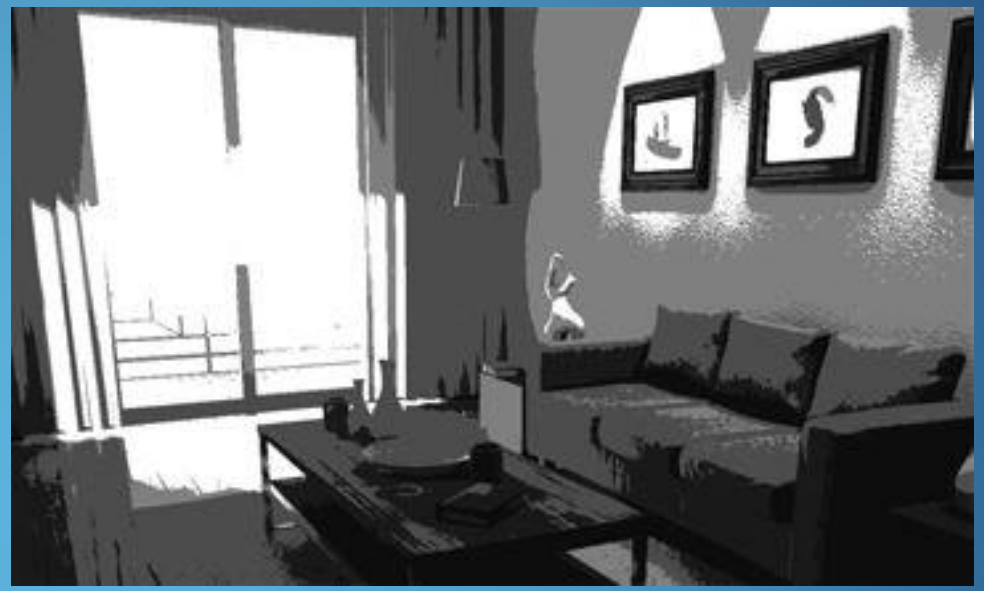
Post Process Effects



Post Process Effects allow artists and designers to tweak the overall look and feel of the scene. Examples include bloom, tone mapping, Sepia.



Post Process Effects



Introduction to Game Development – 2

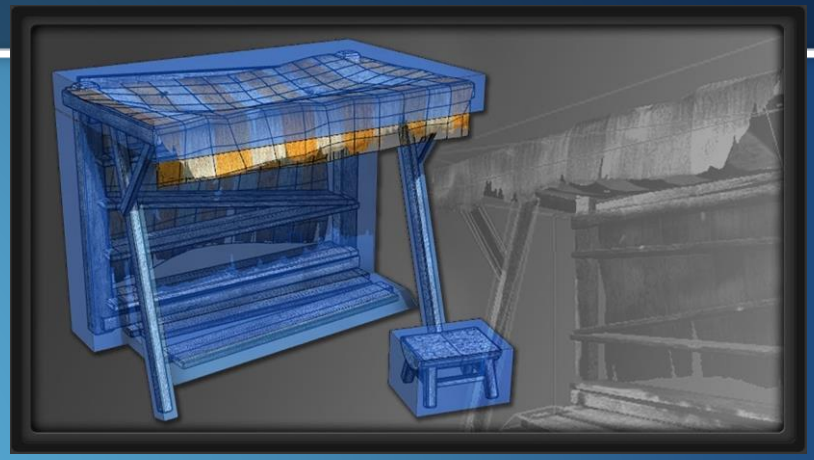
The Development Flow

Part 2 : Physics

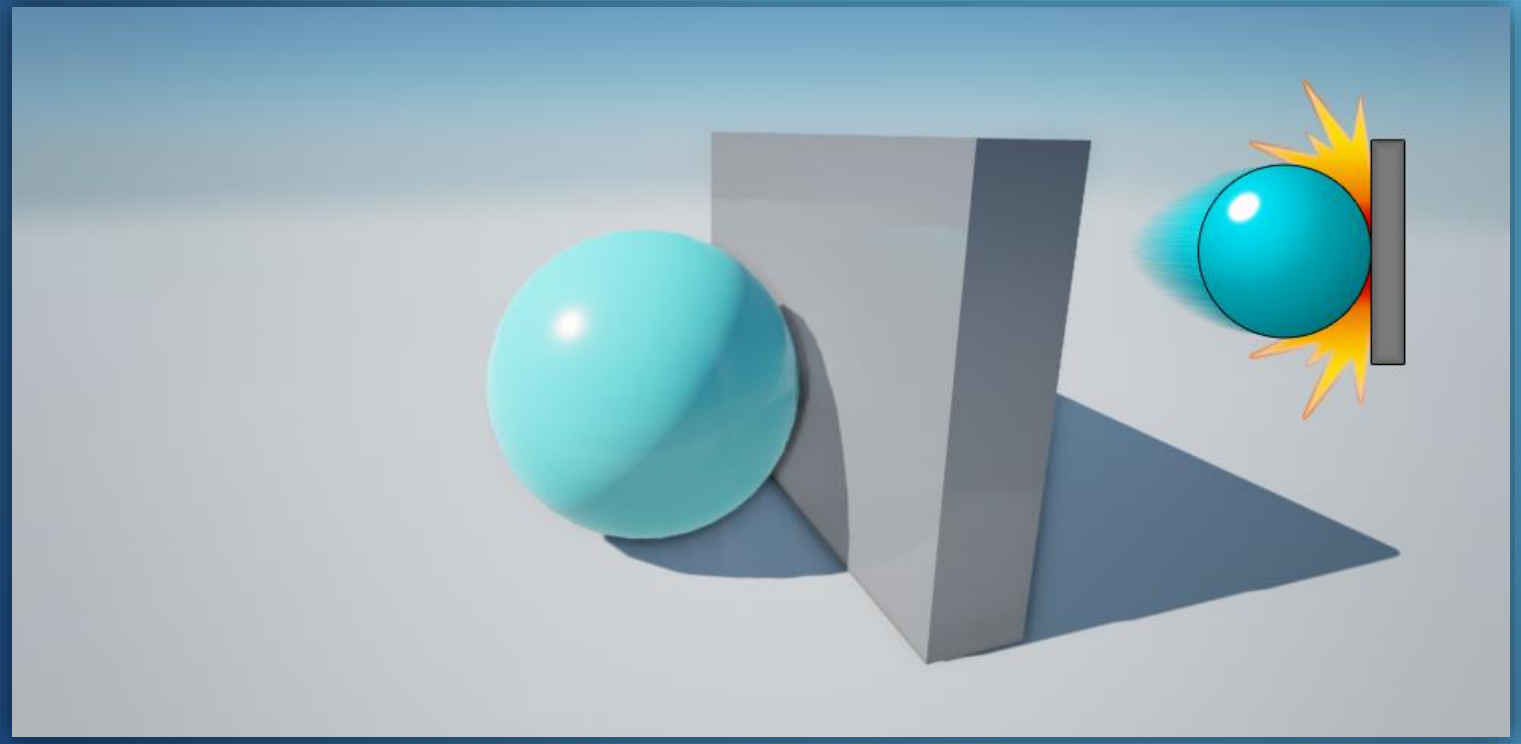
NVIDIA PhysX

NVIDIA PhysX is a scalable multi-platform game physics solution supporting a wide range of devices, from smartphones to high-end multicore CPUs and GPUs. The PhysX SDK provides real time collision detection and simulation of rigid bodies, cloth and fluid particle systems.

Collisions



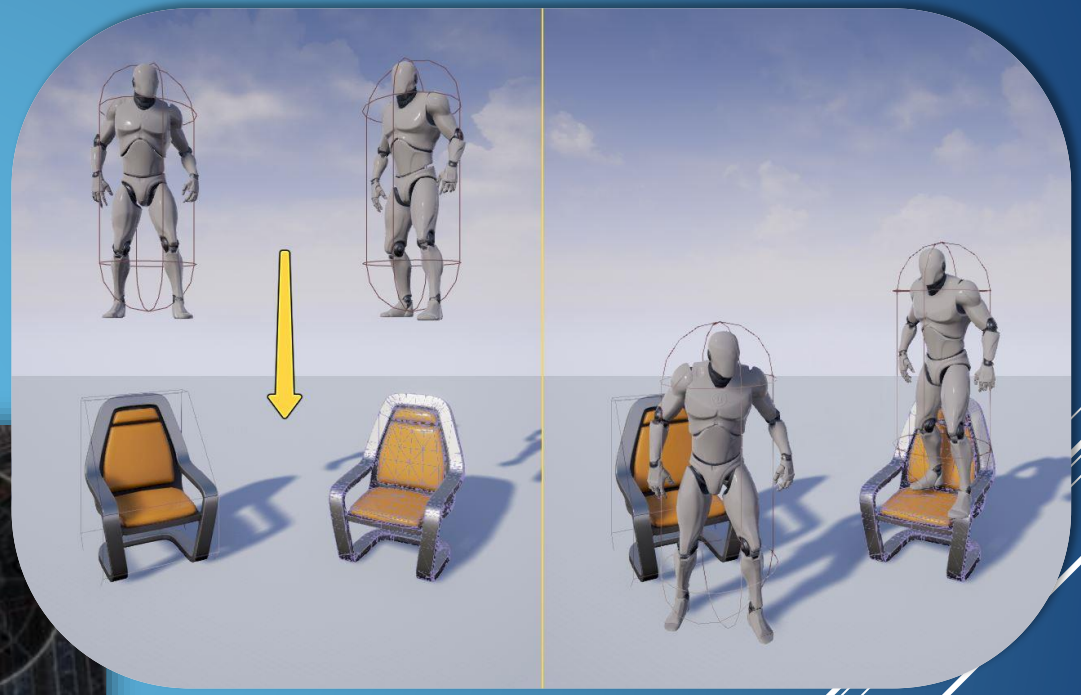
Collision Responses form the basis for how a game engine handles collision during run time. Every object that can collide gets a series of responses that define how it interacts will all other objects. When a collision or overlap event occurs, both (or all) objects involved can be set to affect or be affected by blocking, overlapping, or ignoring each other.

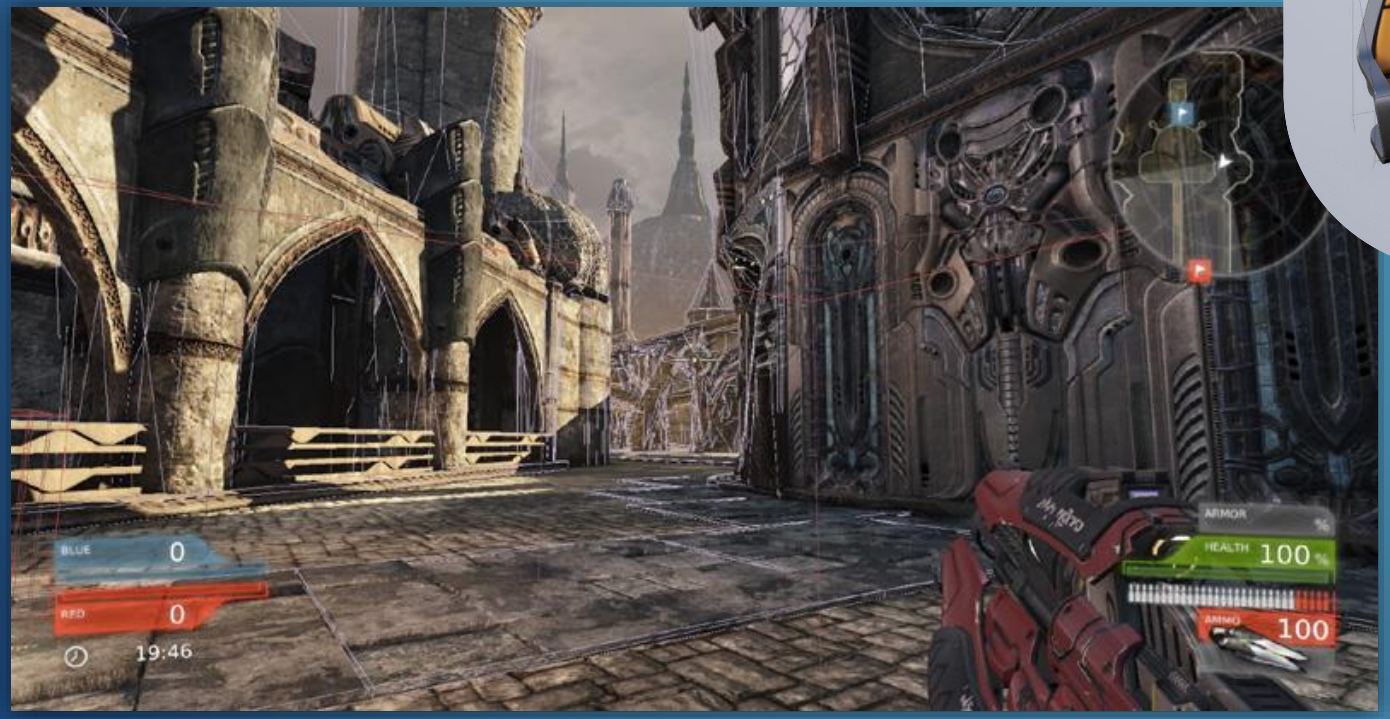


Collisions

Simple vs Complex Collisions

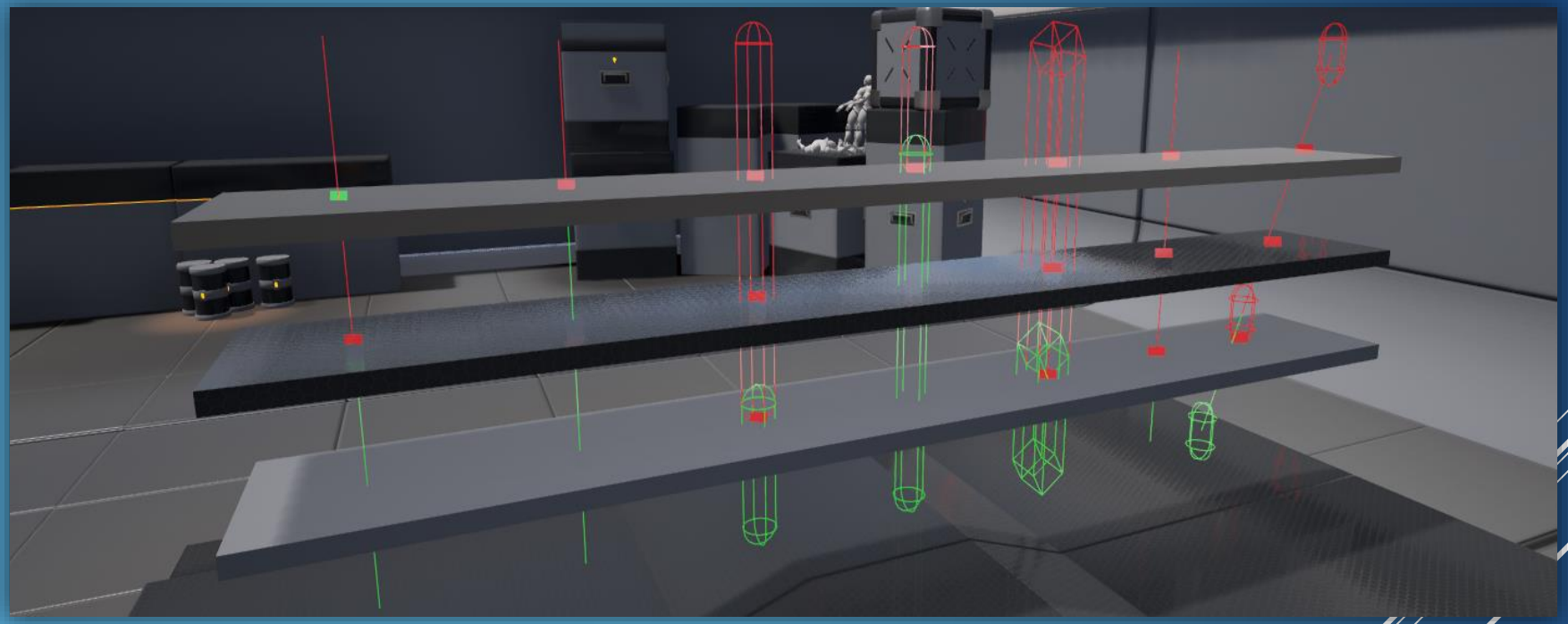
Simple Collisions are primitives like boxes, spheres, capsules. Complex Collisions are the meshes of given objects. Each solid object in your game should have its own collision set up.





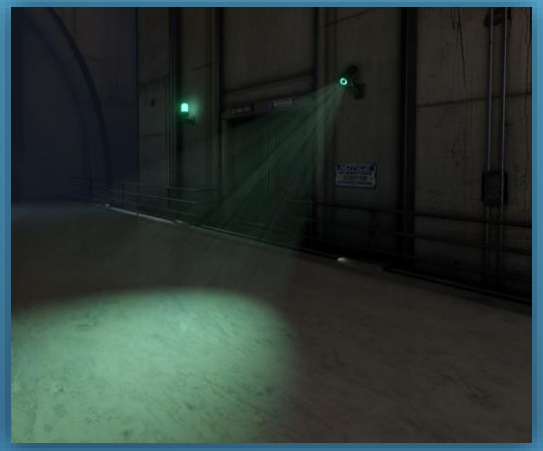
Traces(RayCast)

Traces offer a method for reaching out in your levels and getting feedback on what is present along a line. You use them by providing two end points (a start and end location) and the physics system "traces" a line between those points reporting any object(with collision) that it hits.

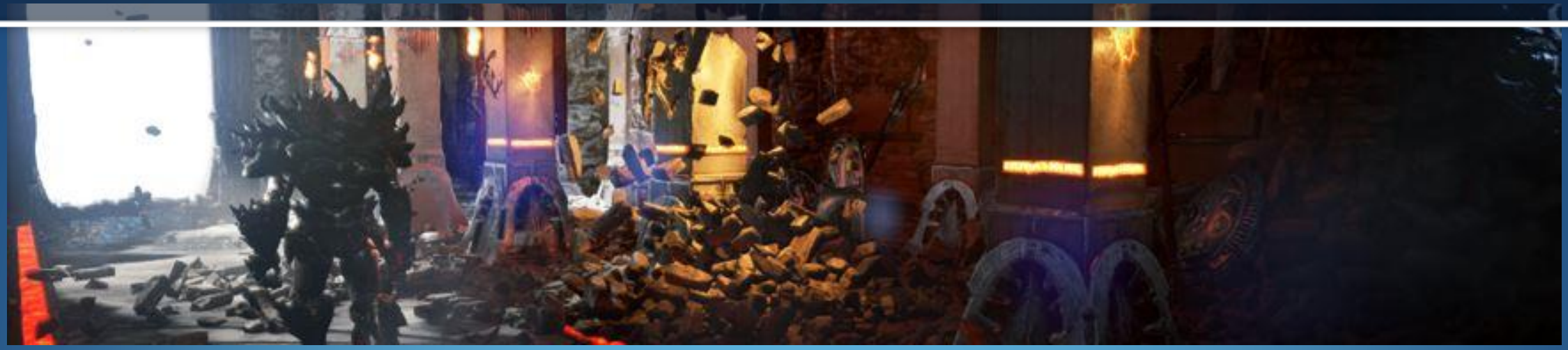


RayCasting

RayCasting is the process of shooting an invisible ray from a point, in a specified direction to detect whether any solid object lay in the path of the ray.



NVIDIA APEX

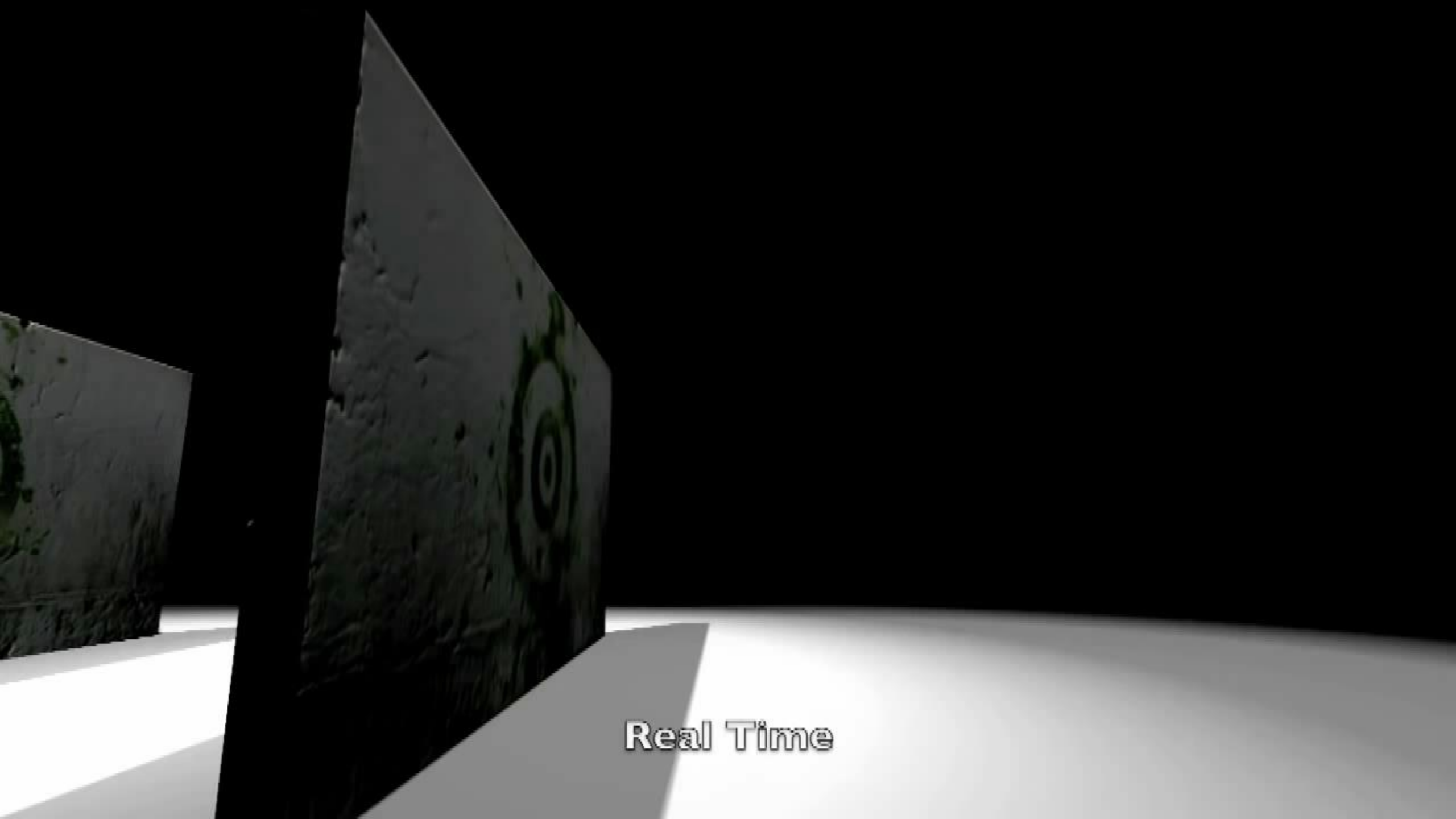


NVIDIA has created a tool called the APEX PhysX Lab, which is a multi-platform, scalable dynamics framework, which puts the artist into the driver seat to quickly create dynamic interactive content.

NVIDIA APEX

APEX Destruction

APEX Destruction enables artists to quickly generate pervasive destruction, which significantly enhances the gaming experience. PhysX Destruction provides support for full and partial destruction and both of these Destruction types can be gameplay affecting or not gameplay affecting. Regardless of the Destruction type you use, both will enhance realism in your game.


Real Time

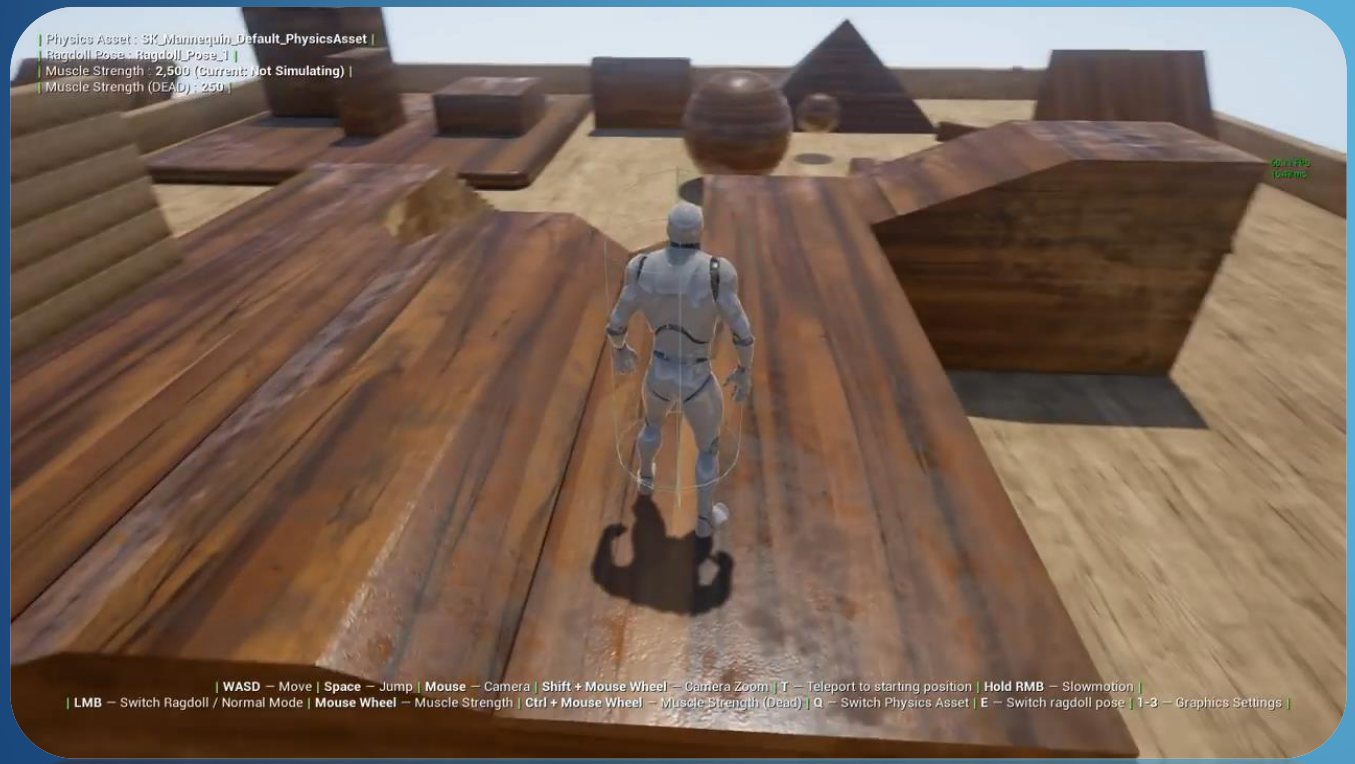
NVIDIA APEX

APEX Clothing

APEX Clothing lets artists quickly generate characters with dynamic clothing to create an ultra realistic interactive gaming experience. PhysX Clothing is artist focused and the authoring tool is intuitive and easy to use. This feature ensures that artists can quickly generate fully simulated and realistic clothing without an extensive knowledge of the underlying PhysX SDK.

Ragdoll Physics

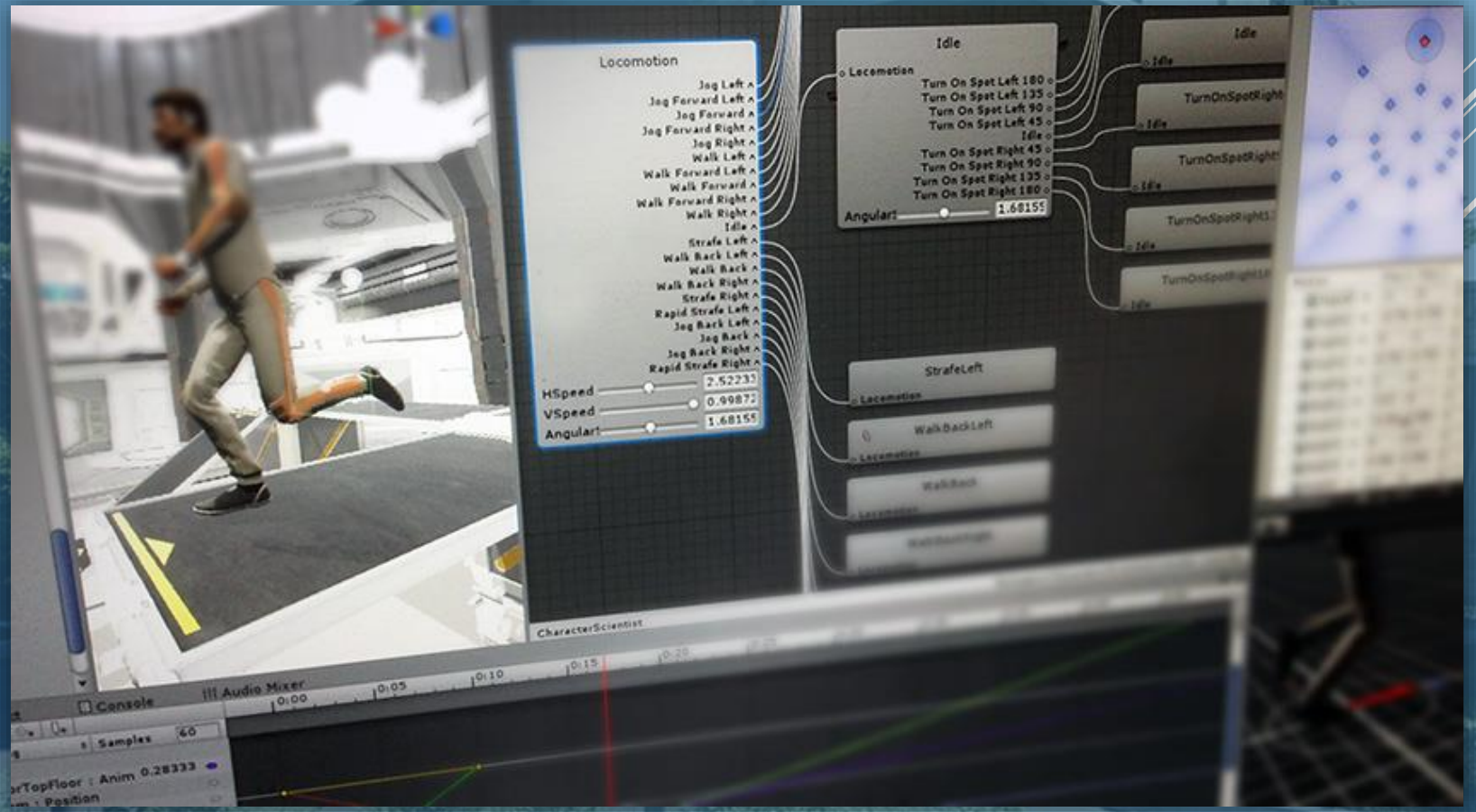
Ragdoll physics is a procedural physics based animation that is often used to replace static death animations in video games.



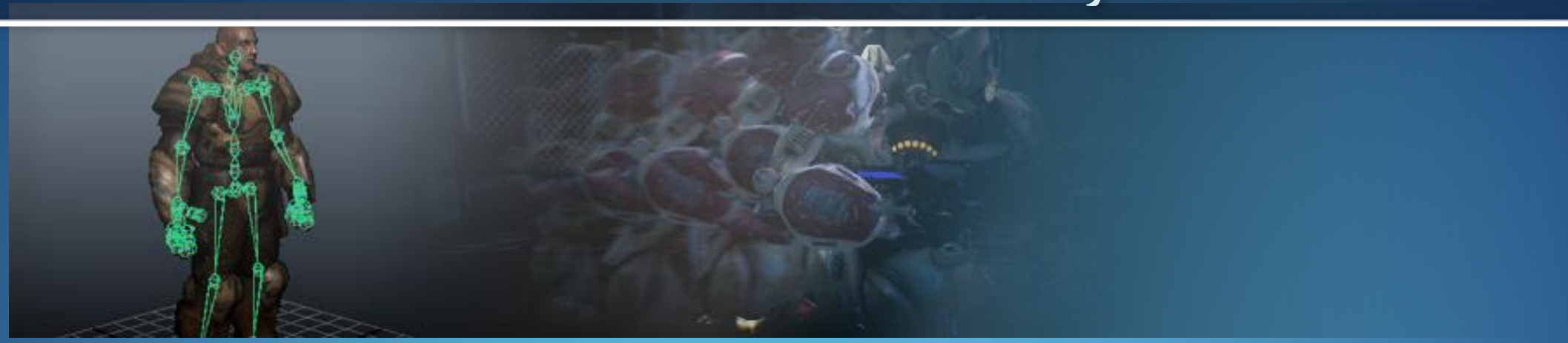
Introduction to Game Development – 2

The Development Flow

Part 3 : Animation System



Skeletal Mesh Animation System



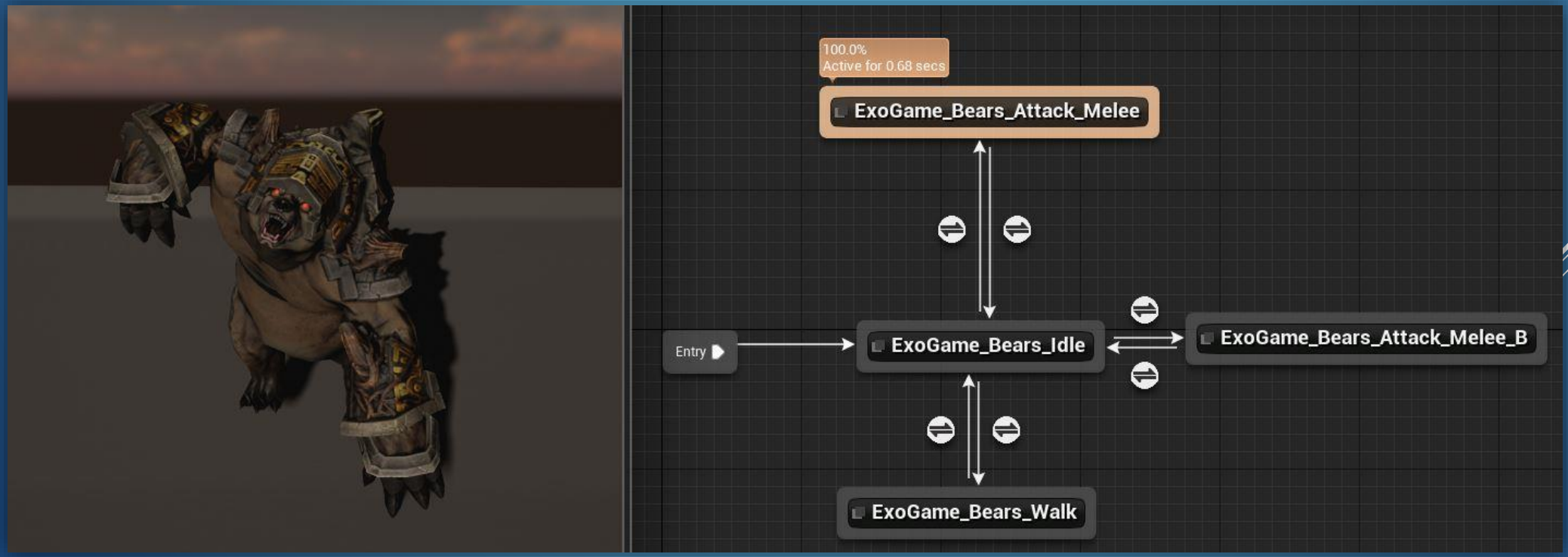
The animation system in a game engine is usually comprised of several Animation Tools and Editors which mixes skeletal-based deformation of meshes with morph-based vertex deformation to allow for complex animation.

This system can be used to make basic player movement appear much more realistic by playing and blending between Animation Sequences, create customized special moves such as scaling ledges and walls , apply damage effects or facial expressions through Morph Targets, directly control the transformations of bones using Skeletal Controls, or create logic based system that determine which animation a character should use in a given situation.

Skeletal Mesh Animation System

State Machines

State Machines provide a graphical way to break the animation of a Skeletal Mesh into a series of States. These states are then governed by Transition Rules that control how to blend from one state to another. As a tool, they greatly simplify the design process for Skeletal Mesh animation, in that you can create a graph that easily controls how your characters can flow between types of animation without having to create a complex Blueprint network.



Skeletal Mesh Animation System

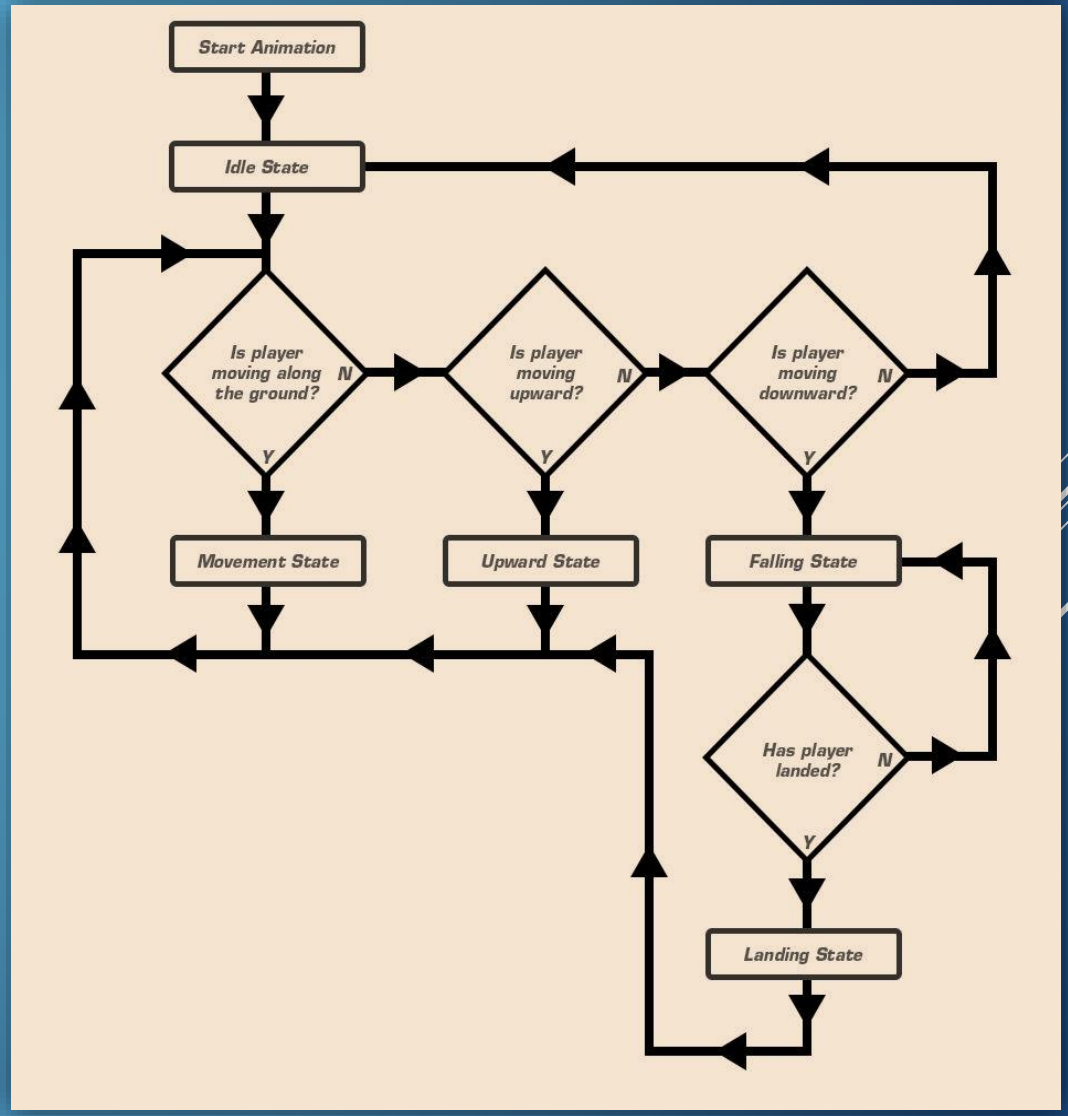
State Machines

States

The basic idea of a state machine is that a character is engaged in some particular kind of action at any given time. The actions available will depend on the type of gameplay but typical actions include things like idling, walking, running, jumping, etc. These actions are referred to as states, in the sense that the character is in a "state" where it is walking, standing still or running.

Transitions

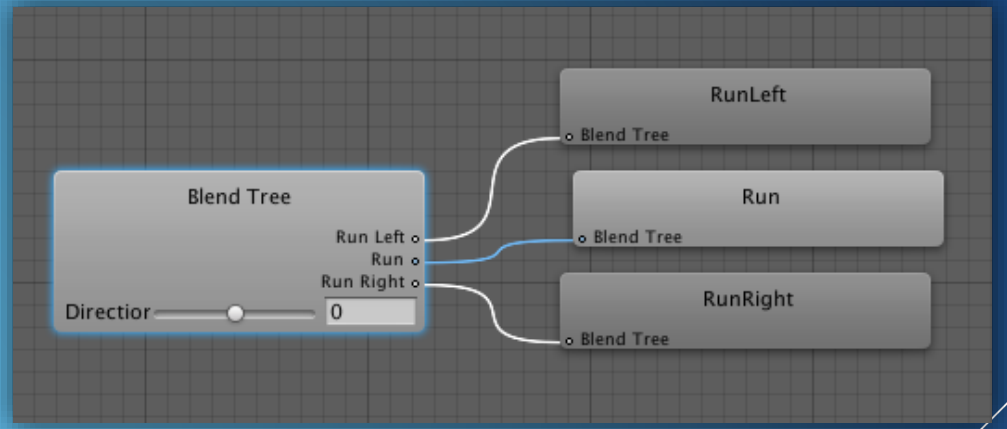
In general, the character will have restrictions on the next state it can go to rather than being able to switch immediately from any state to any other. The options for the next state that a character can enter from its current state are referred to as state transitions.

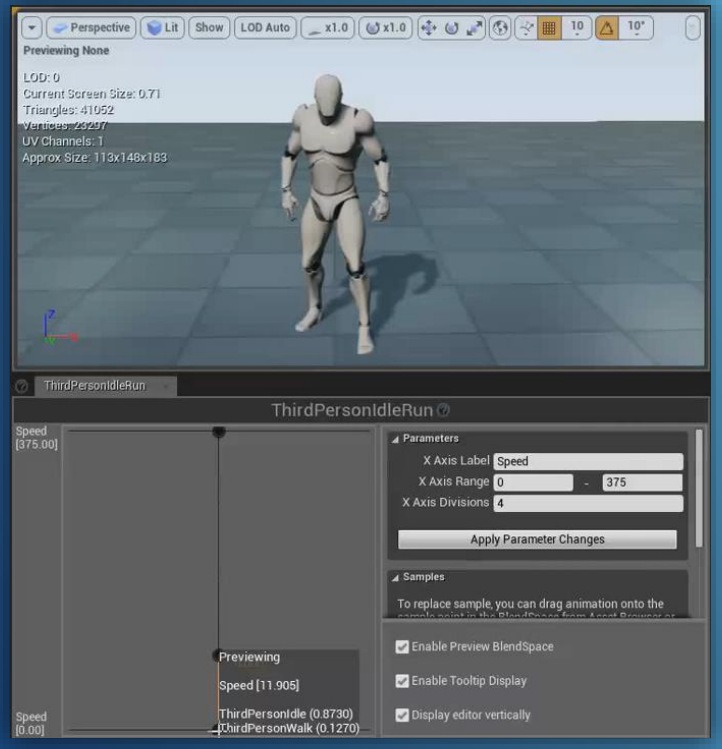


Skeletal Mesh Animation System

Animation Blending

A common task in game animation is to blend between two or more similar motions. Perhaps the best known example is the blending of walking and running animations according to the character's speed. Another example is a character leaning to the left or right as it turns during a run.





Transitions vs Blending

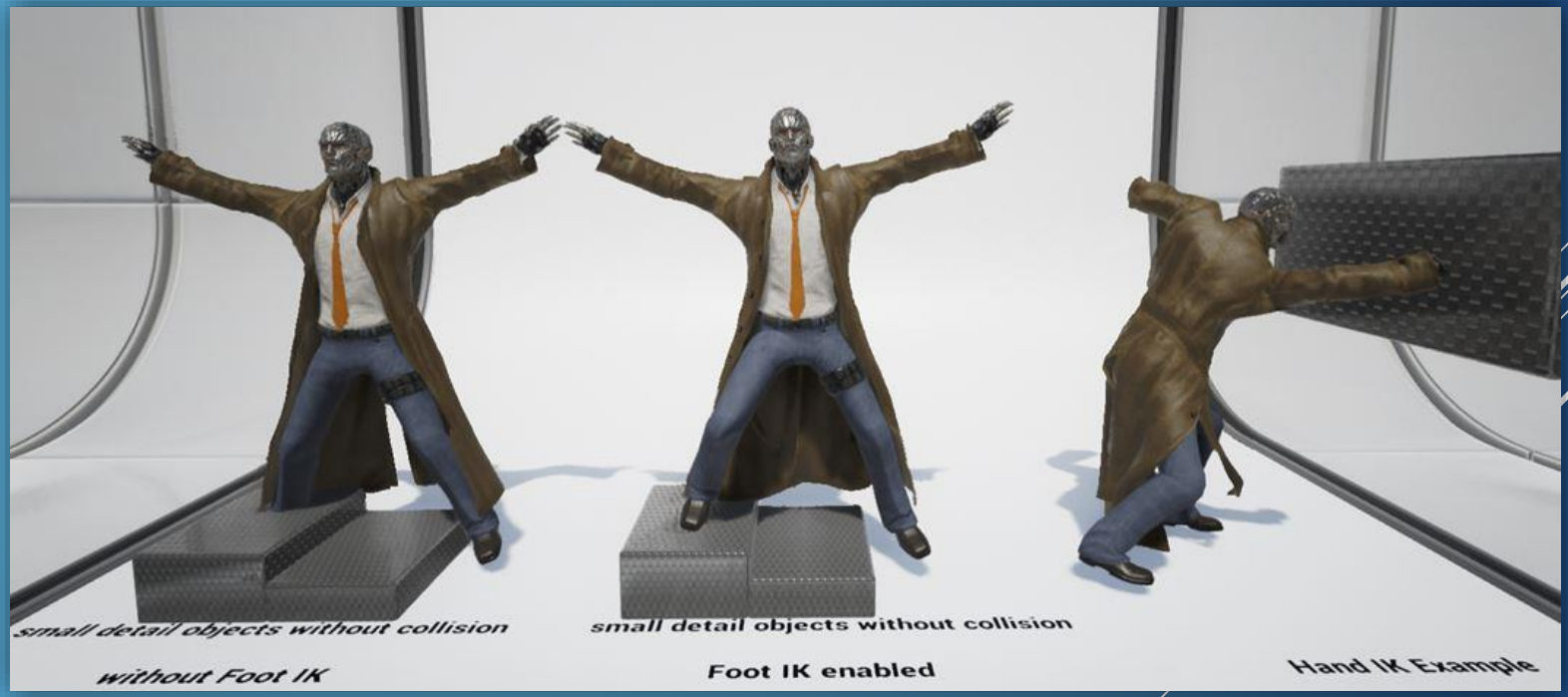
Transitions are used for transitioning smoothly from one Animation State to another over a given amount of time. A transition from one motion to a completely different motion is usually fine if the transition is quick.

Blending is used for allowing multiple animations to be blended smoothly by incorporating parts of them all to varying degrees., The amount that each of the motions contributes to the final effect is controlled using a *blending parameter*. In order for the blended motion to make sense, the motions that are blended must be of similar nature and timing.

Skeletal Mesh Animation System

Inverse Kinematics

Inverse Kinematics (IK) provide a way to handle joint rotation from the location of an end-effector rather than via direct joint rotation. In practice, you provide an effector location and the IK solution then solves the rotation so that the final joint coincides with that location as best it can. This can be used to keep a character's feet planted on uneven ground, and in other ways to produce believable interactions with the world.



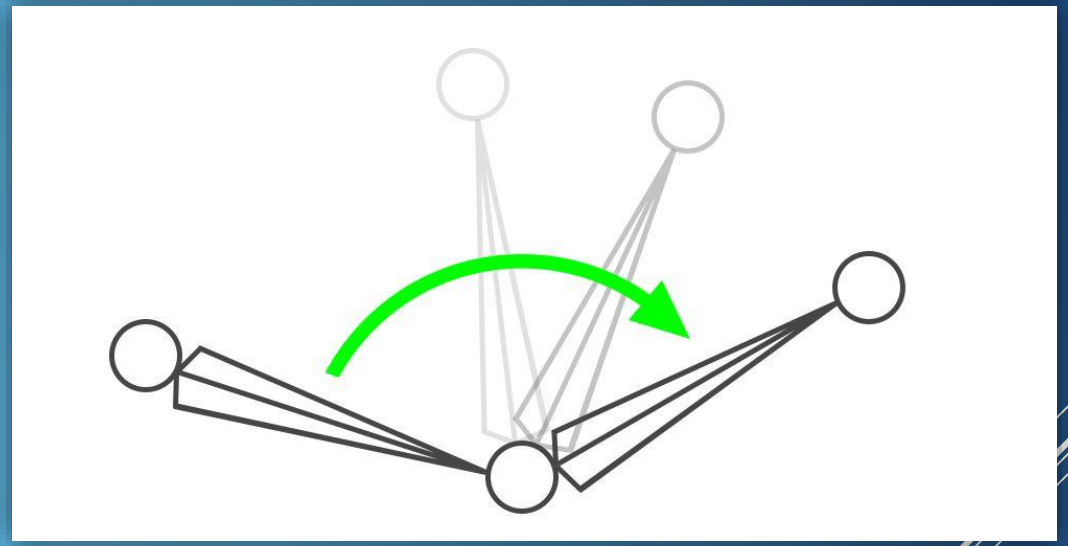
Skeletal Mesh Animation System

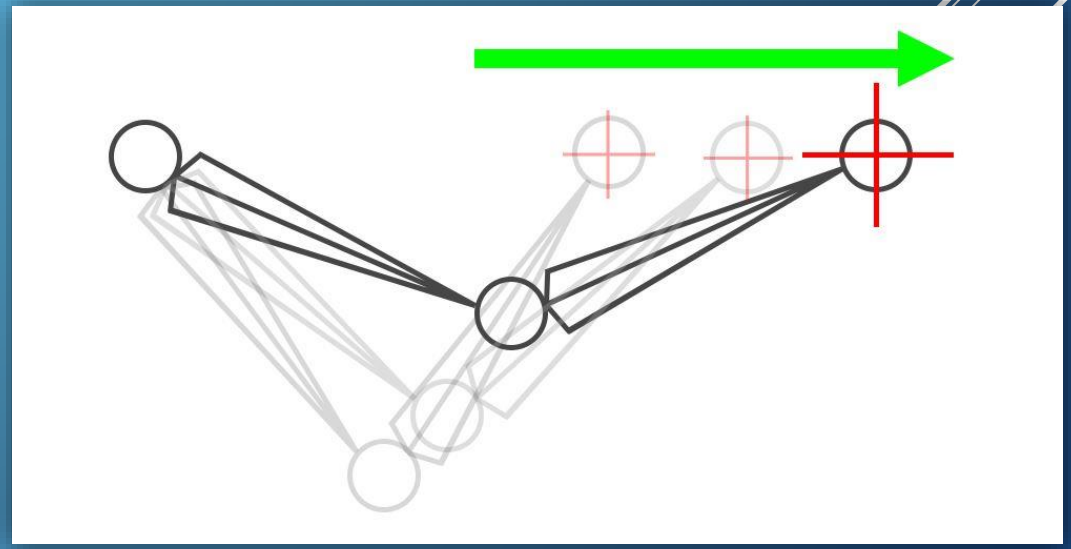
Inverse Kinematics

FK vs IK

Most animated skeletons are driven by direct rotational data fed straight into the bones of the character or Skeletal Mesh. This can be thought of as forward kinematics, or direct application of rotation to joints or bones.

Inverse kinematics works in the other direction. Instead of applying rotation to bones, we instead give the bone chain a target (also known as an end effector), providing a position that the end of the chain should try to achieve. The user or animator moves the effector and the IK solver (the algorithm that drives rotation in an IK system) rotates the bones such that the final bone in the chain ends at the location of the target.





Skeletal Mesh Animation System

Root Motion

Root Motion is the motion of a character that is based off animation from the root bone of the skeleton.

Typically in game animation, a character's collision capsule is driven through the scene by the controller. Data from this capsule is then used to drive animation. For instance, if the capsule is moving forward, the system then knows to play a running or walking animation on the character to give the appearance that the character is moving under its own power. However, in some cases, it makes sense for complex animations to actually drive the collision capsule. This is where Root Motion handling becomes critical for your games.

Introduction to Game Development – 2

The Development Flow

Part 4 : Navigation and Pathfinding

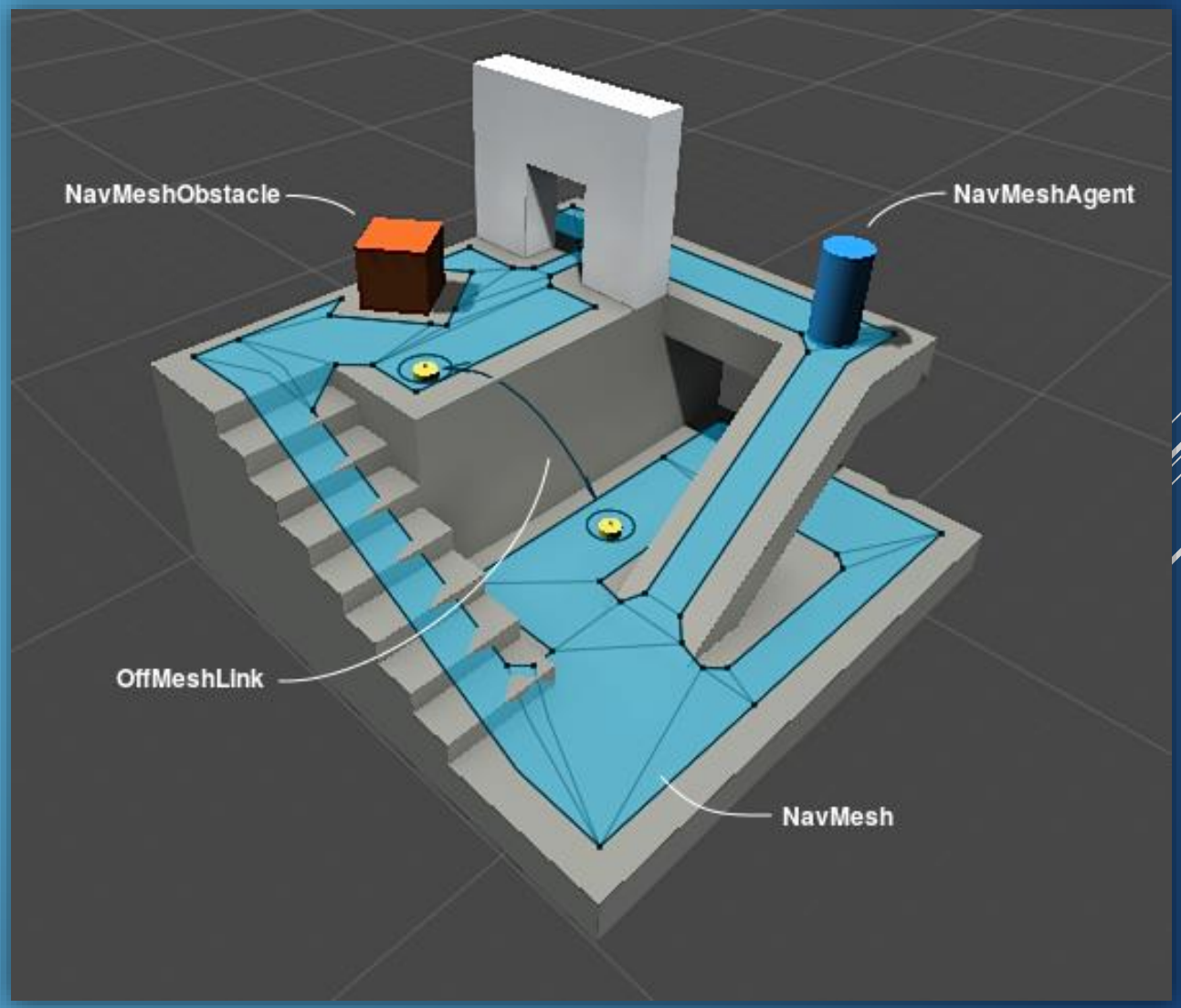
The navigation system allows you to create characters that can intelligently move around the game world, using navigation meshes that are created automatically from your Scene geometry. Dynamic obstacles allow you to alter the navigation of the characters at runtime.

Navigation System Overview

The Navigation System allows you to create characters which can navigate the game world. It gives your characters the ability to understand that they need to take stairs to reach second floor, or to jump to get over a ditch.

NavMesh

NavMesh (short for Navigation Mesh) is a data structure which describes the walkable surfaces of the game world and allows to find path from one walkable location to another in the game world. The data structure is built, or baked, automatically from your level geometry.

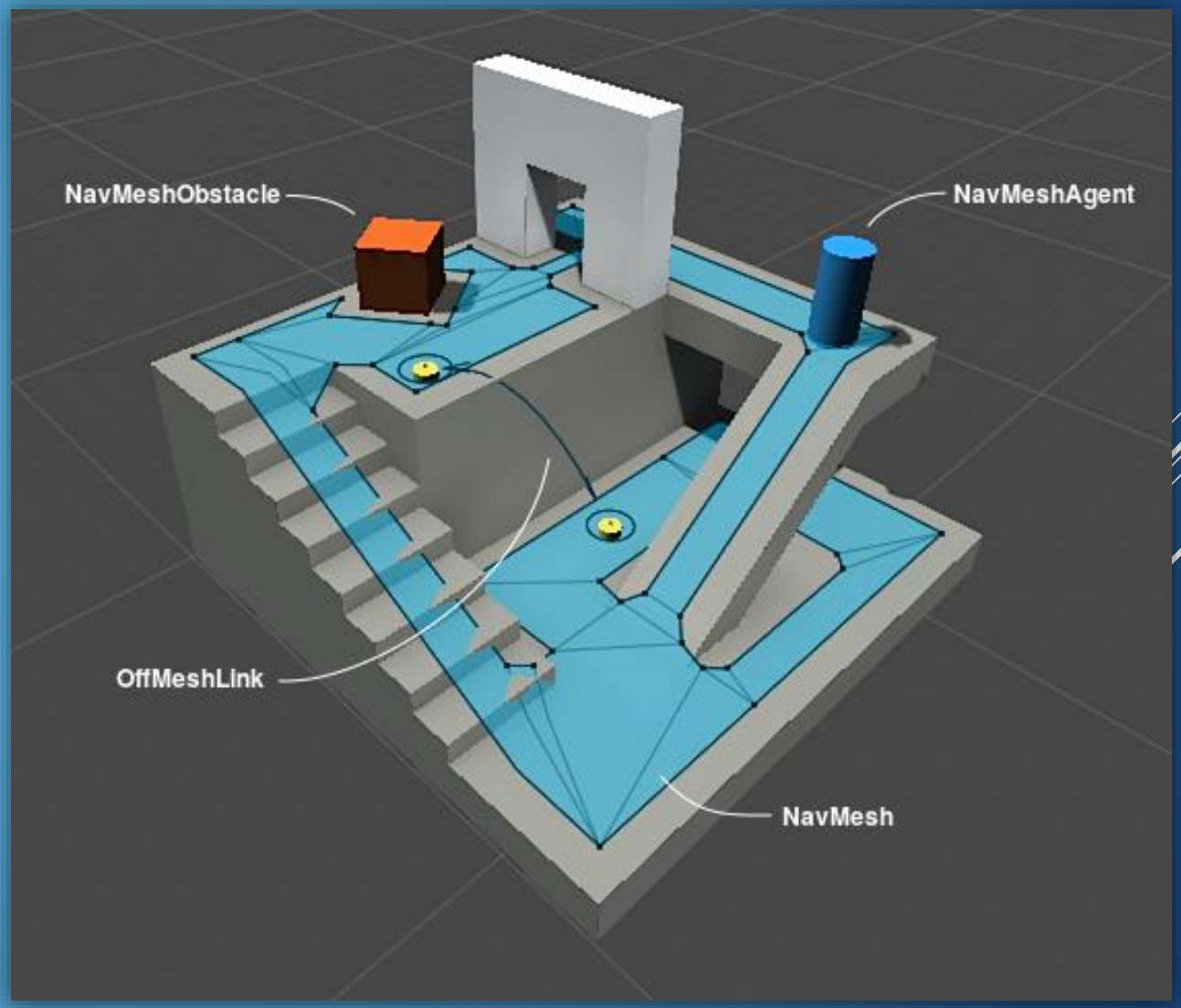


Navigation System Overview

The Navigation System allows you to create characters which can navigate the game world. It gives your characters the ability to understand that they need to take stairs to reach second floor, or to jump to get over a ditch.

NavMesh Agent

NavMesh Agent helps you to create characters which avoid each other while moving towards their goal. Agents reason about the game world using the NavMesh and they know how to avoid each other as well as moving obstacles.

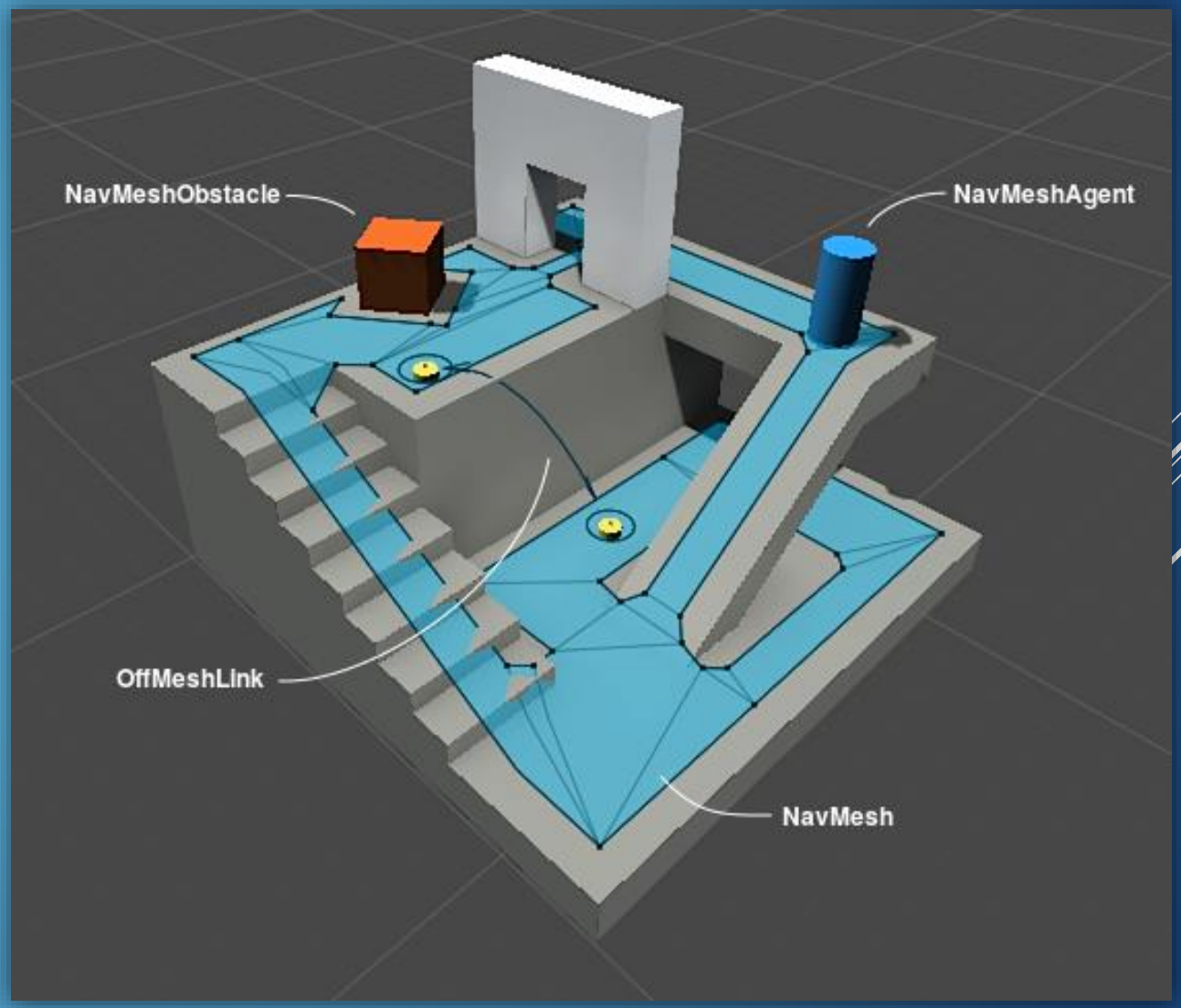


Navigation System Overview

The Navigation System allows you to create characters which can navigate the game world. It gives your characters the ability to understand that they need to take stairs to reach second floor, or to jump to get over a ditch.

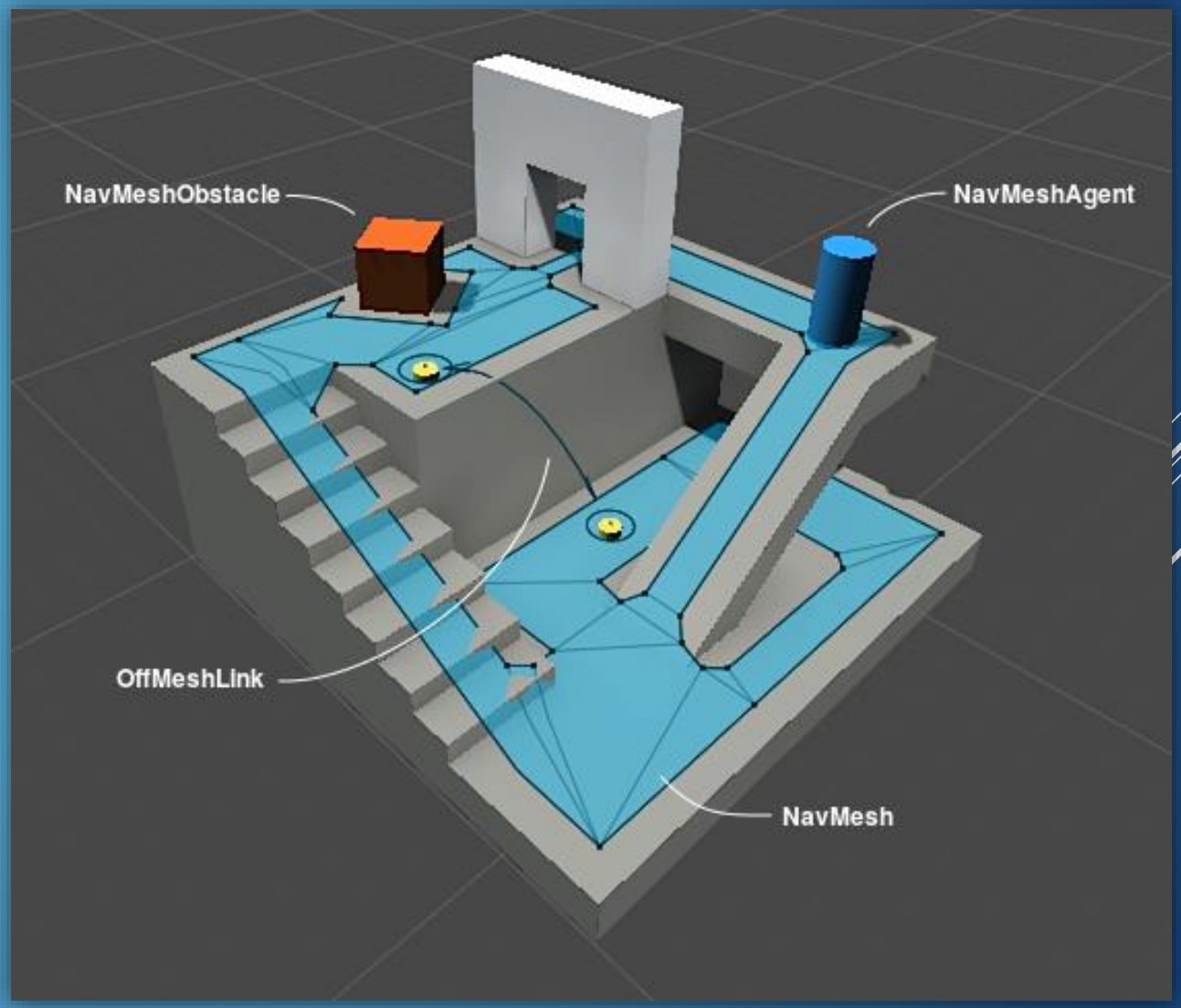
Off-Mesh Link

Off-Mesh Link allows you to incorporate navigation shortcuts which cannot be represented using a walkable surface. For example, jumping over a ditch or a fence, or opening a door before walking through it, can be all described as Off-mesh links.



Navigation System Overview

The Navigation System allows you to create characters which can navigate the game world. It gives your characters the ability to understand that they need to take stairs to reach second floor, or to jump to get over a ditch.



NavMesh Obstacle

NavMesh Obstacle allows you to describe moving obstacles the agents should avoid while navigating the world. While the obstacle is moving the agents do their best to avoid it, but once the obstacle becomes stationary it will carve a hole in the NavMesh so that the agents can change their paths to steer around it, or if the stationary obstacle is blocking the path way, the agents can find a different route.

Inner Workings of the Navigation System

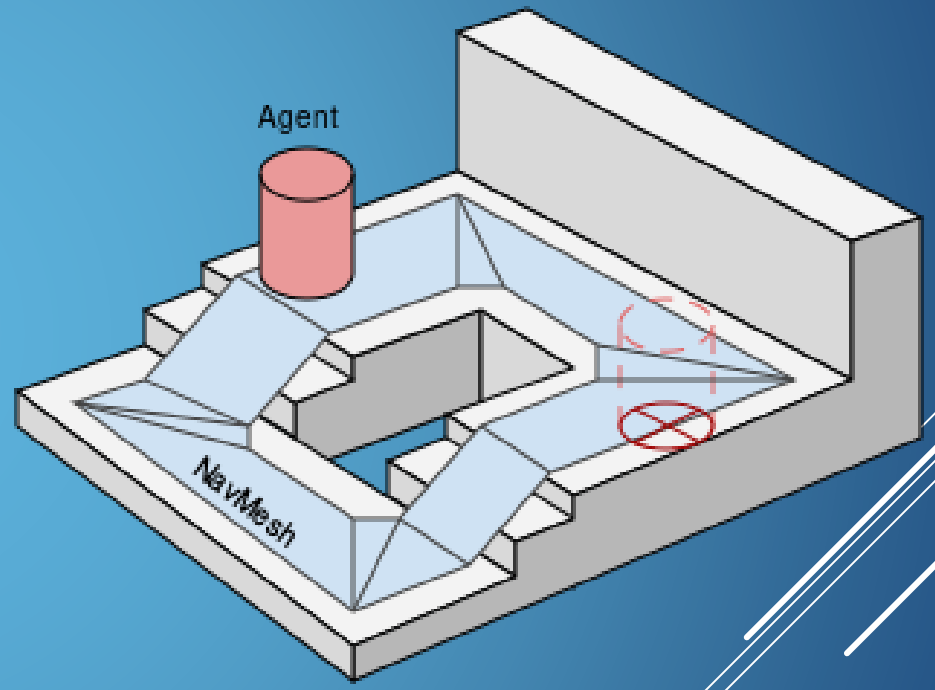
When you want to intelligently move characters in your game (or agents as they are called in AI circles), you have to solve two problems: how to *reason* about the level to find the destination, then how to *move* there. These two problems are tightly coupled, but quite different in nature. The problem of reasoning about the level is more global and static, in that it takes into account the whole scene. Moving to the destination is more local and dynamic, it only considers the direction to move and how to prevent collisions with other moving agents.

Inner Workings of the Navigation System

Walkable Areas

The walkable areas define the places in the scene where the agent can stand and move. The walkable area is built automatically from the geometry in the scene by testing the locations where the agent can stand. Then the locations are connected to NavMesh.

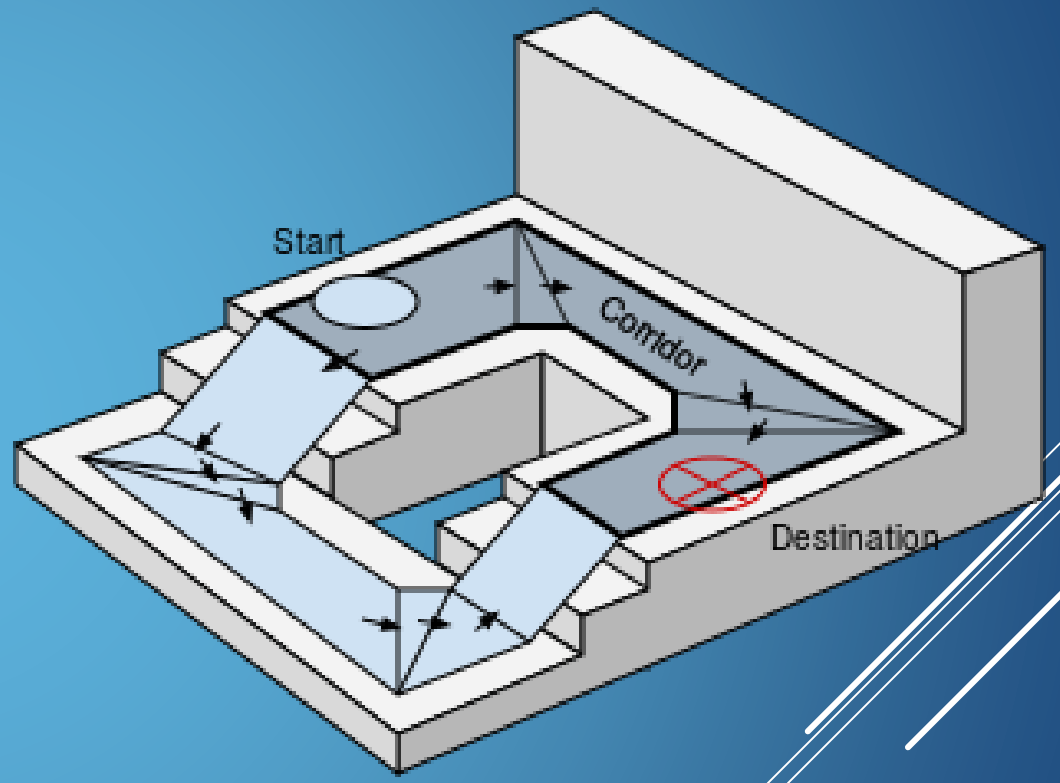
The NavMesh stores this surface as convex polygons. Convex polygons are a useful representation, since we know that there are no obstructions between any two points inside a polygon. In addition to the polygon boundaries, we store information about which polygons are neighbors to each other. This allows us to reason about the whole walkable area.



Inner Workings of the Navigation System

Finding Paths

To find path between two locations in the scene, we first need to map the start and destination locations to their nearest polygons. Then we start searching from the start location, visiting all the neighbors until we reach the destination polygon. Tracing the visited polygons allows us to find the sequence of polygons which will lead from the start to the destination. A common algorithm to find the path is A* (pronounced “A star”).

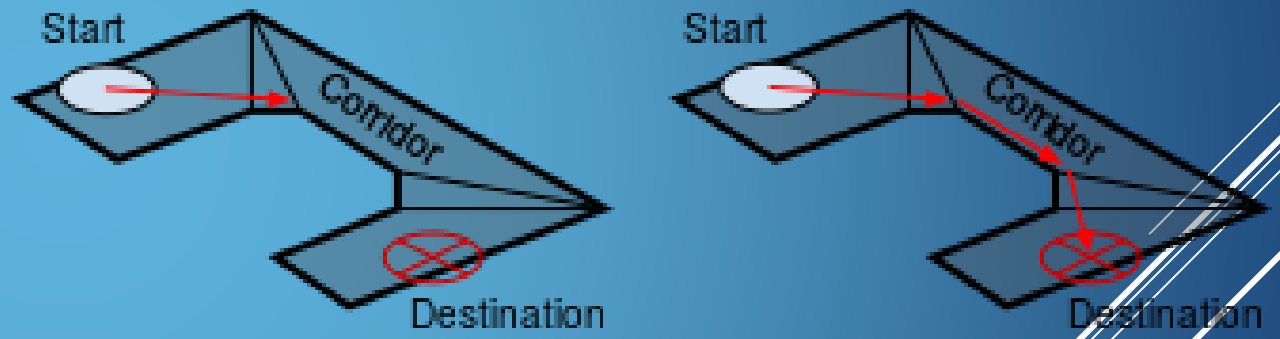


Inner Workings of the Navigation System

Following the Path

The sequence of polygons which describe the path from the start to the destination polygon is called a corridor. The agent will reach the destination by always steering towards the next visible corner of the corridor. When dealing with multiple agents moving at the same time, they will need to deviate from the original path when avoiding each other. Trying to correct such deviations using a path consisting of line segments soon becomes very difficult and error prone.

Since the agent movement in each frame is quite small, we can use the connectivity of the polygons to fix up the corridor in case we need to take a little detour. Then we quickly find the next visible corner to steer towards.

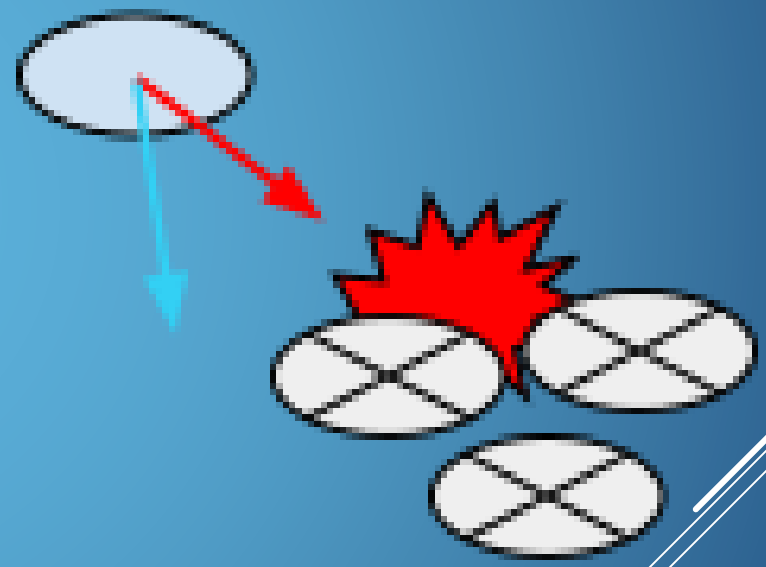


Inner Workings of the Navigation System

Avoiding Obstacles

The steering logic takes the position of the next corner and based on that figures out a desired direction and speed (or velocity) needed to reach the destination.

Obstacle avoidance chooses a new velocity which balances between moving in the desired direction and preventing future collisions with other agents and edges of the navigation mesh.



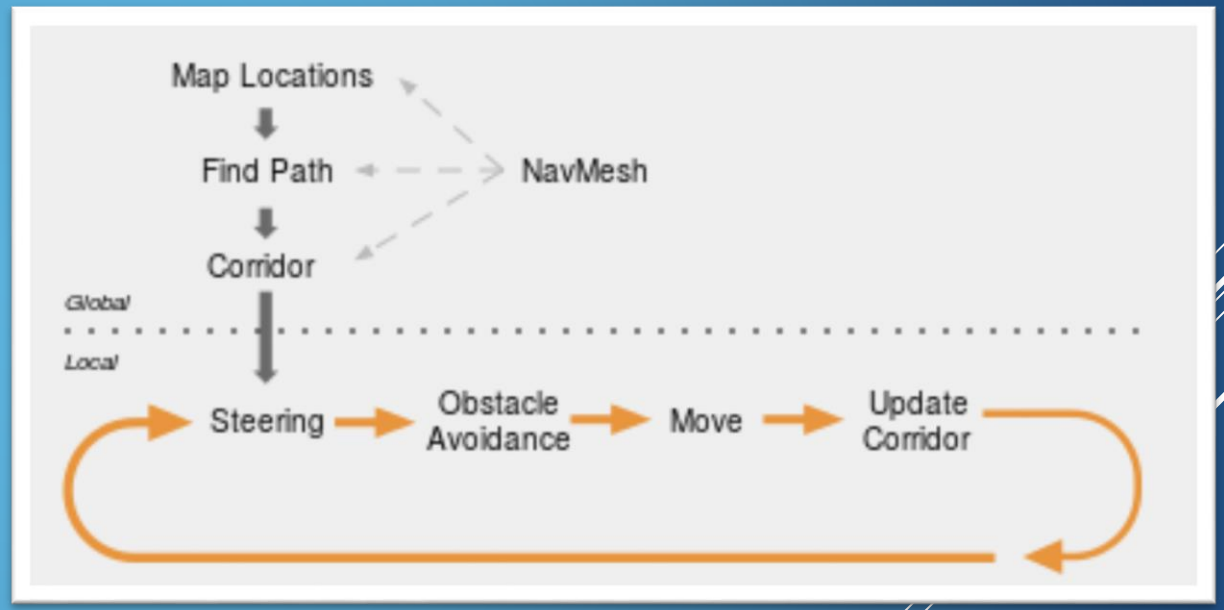
Moving the Agent

Finally after steering and obstacle avoidance the final velocity is calculated. Once the agent has been moved, the simulated agent location is moved and constrained to NavMesh. This last small step is important for robust navigation.

Inner Workings of the Navigation System

Global vs Local Navigation

Global navigation is used to find the corridor across the world. Finding a path across the world is a costly operation requiring quite a lot of processing power and memory.
The linear list of polygons describing the path is a flexible data structure for steering, and it can be locally adjusted as the agent's position moves. Local navigation tries to figure out how to efficiently move towards the next corner without colliding with other agents or moving objects.



Introduction to Game Development – 2

The Development Flow

Part 5 : The Development Pipeline

The Development Pipeline

Story

- Concept Art
- Scripts
- Storyboarding

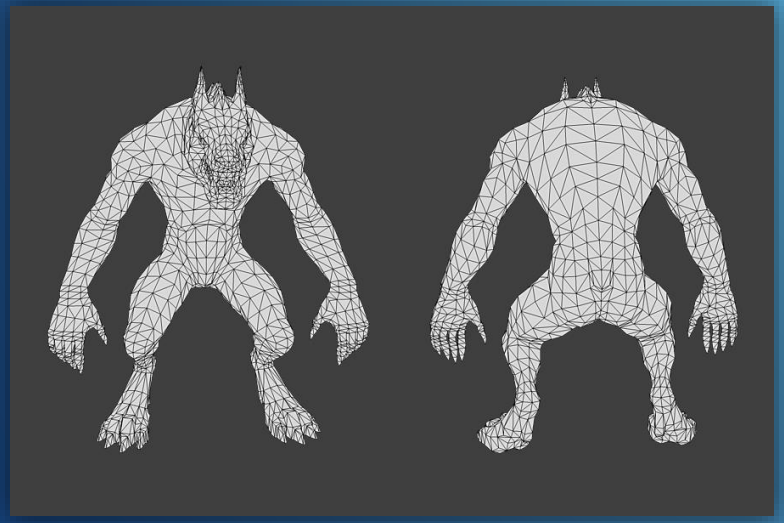
The Development Pipeline

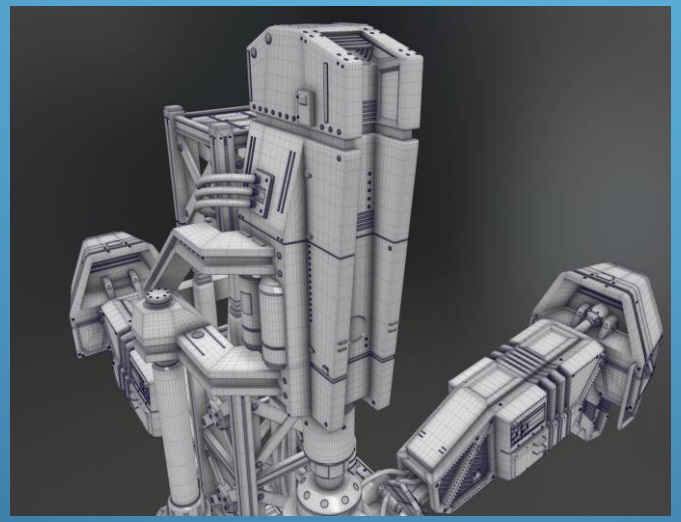
Asset Creation

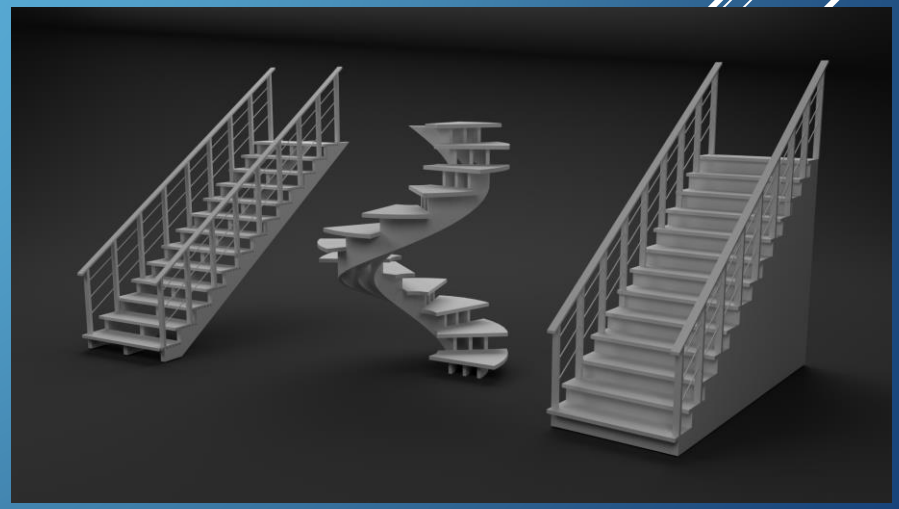
Modelling	Animations	Texturing
• Characters • Props • Sets	• Rigging • Skinning • Animations	• Creating Texture Maps • UV Wrapping • Creating Normal and Lightmap Texture

Modelling

Creating 3D Models of Characters , Props and Environments.



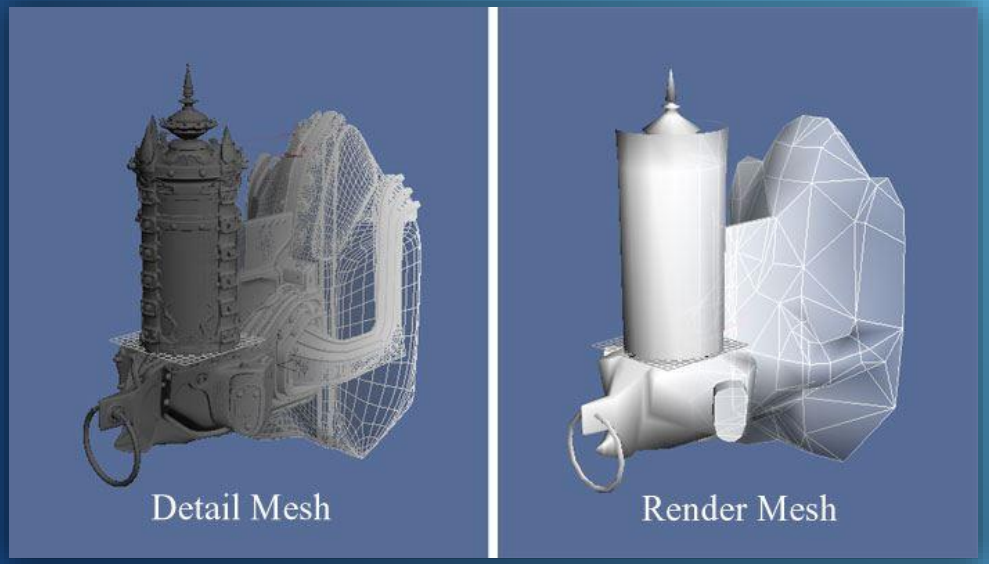




The Development Pipeline

Asset Creation

Modelling	Animations	Texturing
• Characters • Props • Sets	• Rigging • Skinning • Animations	• Creating Texture Maps • UV Wrapping • Creating Normal and Lightmap Texture



Modelling

Make a high Poly version of your model , either in your 3d Modelling software or using sculpting technique in Zbrush or any other preferred Sculpting software. Use this model to bake Normal Maps.

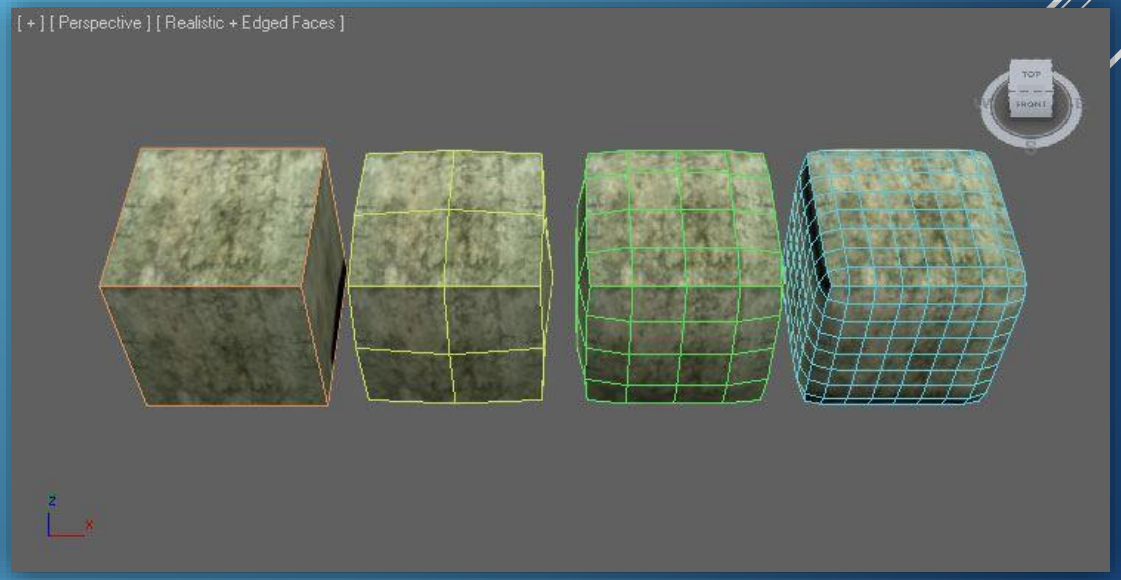
The Development Pipeline

Asset Creation

Modelling	Animations	Texturing
• Characters • Props • Sets	• Rigging • Skinning • Animations	• Creating Texture Maps • UV Wrapping • Creating Normal and Lightmap Texture

Modelling

Create additional Level of Details(LOD) for your mesh.



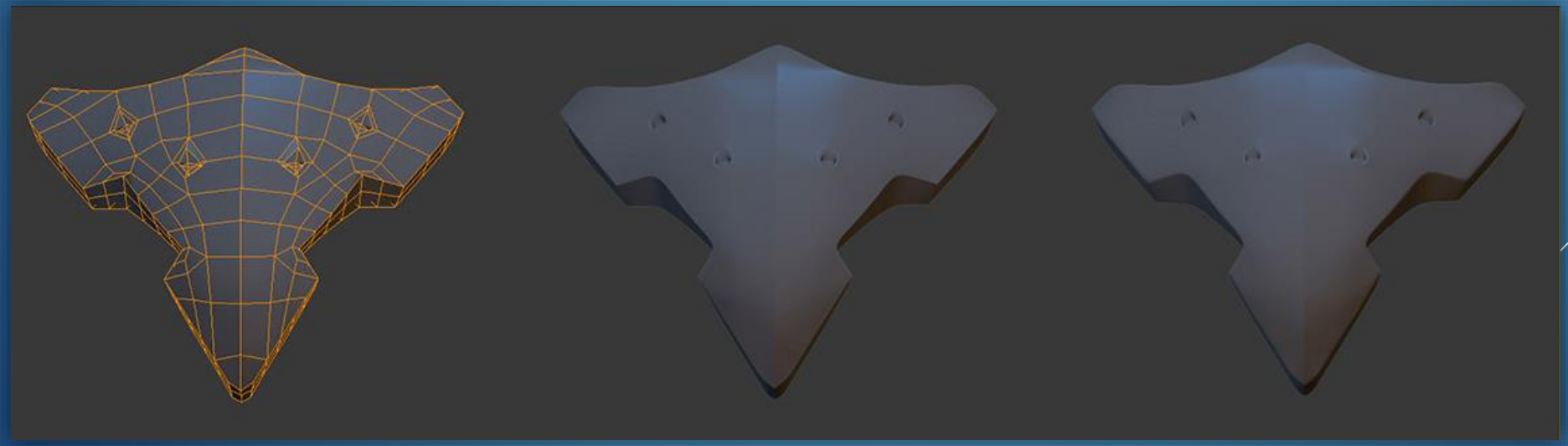
The Development Pipeline

Asset Creation

Modelling	Animations	Texturing
• Characters • Props • Sets	• Rigging • Skinning • Animations	• Creating Texture Maps • UV Wrapping • Creating Normal and Lightmap Texture

Modelling

Define Smoothing Groups for your meshes



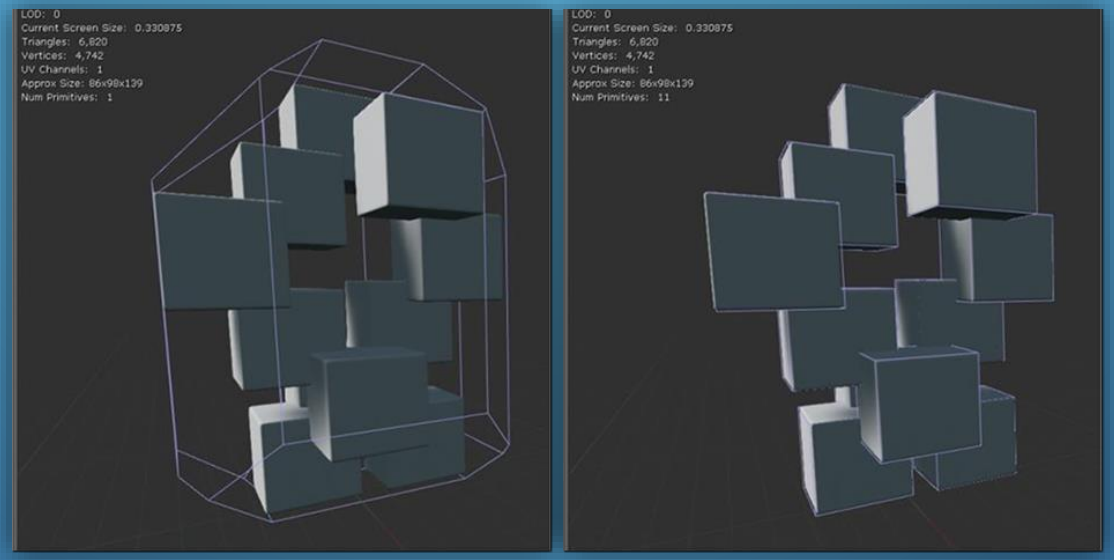
The Development Pipeline

Asset Creation

Modelling	Animations	Texturing
• Characters • Props • Sets	• Rigging • Skinning • Animations	• Creating Texture Maps • UV Wrapping • Creating Normal and Lightmap Texture

Modelling

Create Collision Meshes for your Models



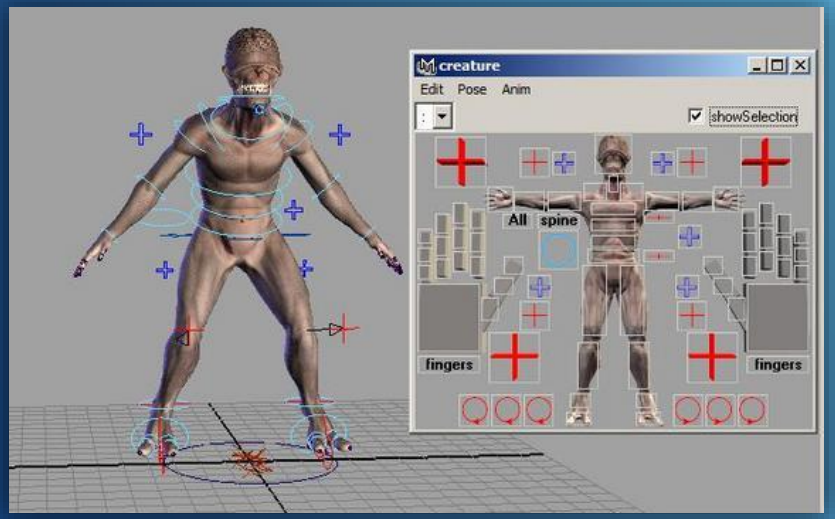
The Development Pipeline

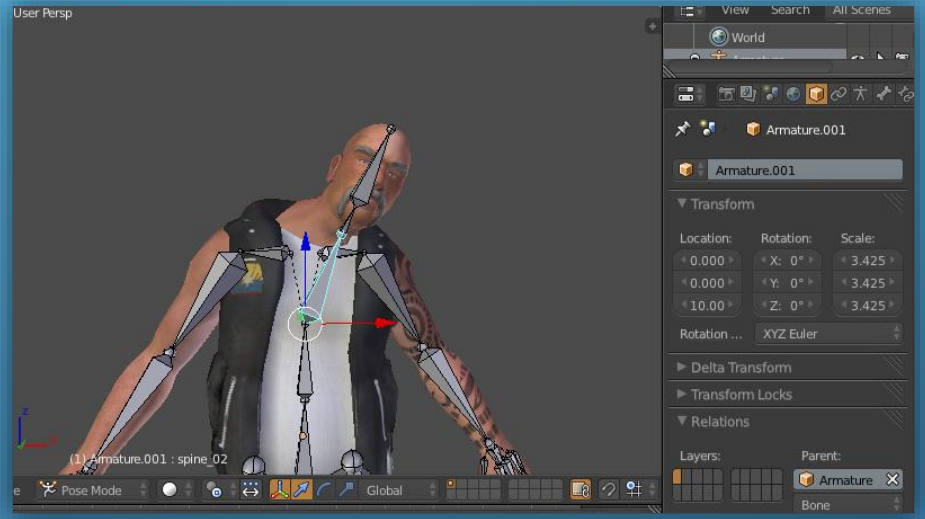
Asset Creation

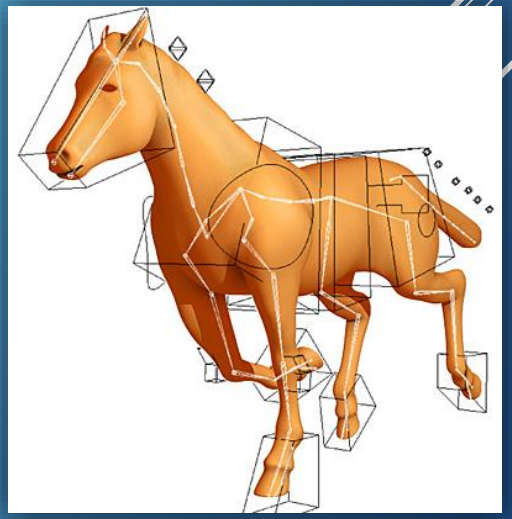
Modelling	Animations	Texturing
• Characters • Props • Sets	• Rigging • Skinning • Animations	• Creating Texture Maps • UV Wrapping • Creating Normal and Lightmap Texture

Animations

Rig and Skin your Organic Characters







The Development Pipeline

Asset Creation

Modelling	Animations	Texturing
• Characters • Props • Sets	• Rigging • Skinning • Animations	• Creating Texture Maps • UV Wrapping • Creating Normal and Lightmap Texture

Animations

Create Animation Sequences for your Characters. Based on the software you use for creating animations, you may create one or multiple animation sequences at a time for the same character.