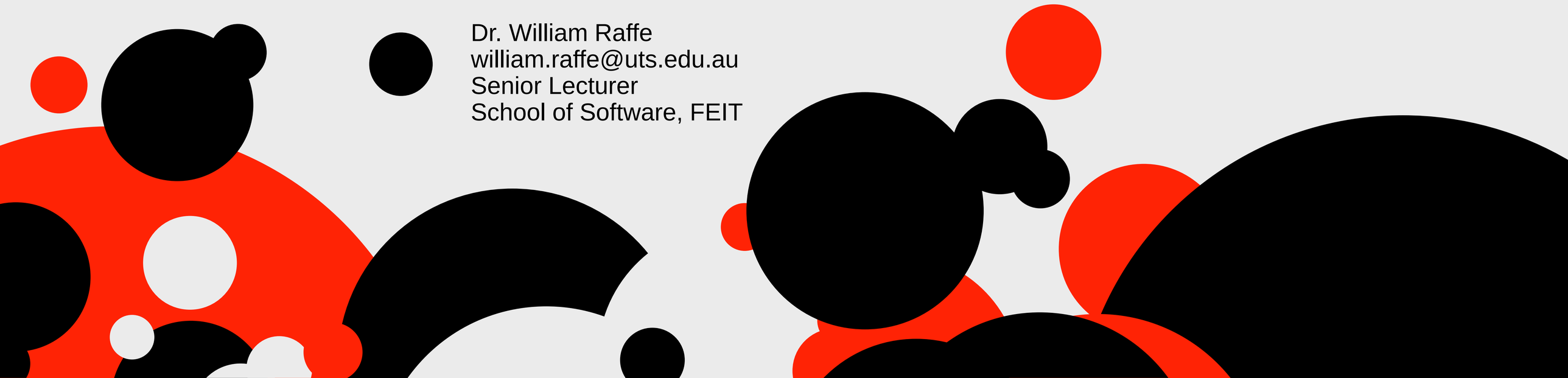
31263 / 32004
Intro to Games Development
Week 2

Dr. William Raffe
william.raffe@uts.edu.au
Senior Lecturer
School of Software, FEIT



Overview

- **Graphics Fundamentals**
- **Camera**
- **Lights**
- **3D Primitives**
- **Renderer Components**
- **Materials**
- **Textures**
- **Sprites**
- **Particle Effects**

Just a taster

- Most of the content in this lecture is covered more so in **31264 / 32501 Introduction to Computer Graphics** where it is covered in more detail and including the theory

- Here it is presented just to get you started with understanding the basics of computer graphics and how they are used in an engine like Unity.

- Many of these concepts are created by **Engine Developers**
  - Need a deep understanding of physics, maths, and human perception to create these techniques from scratch.

- Many of the concepts are used by **Game Developers** and **Digital Artists**
  - To make and polish the visuals of a game or animation

Camera

- A window into the game’s scene that represents the player’s view
  - What the camera sees is what the player sees
  - Where the camera goes is where the player goes

- A camera is just a game object with a Camera component
  - You can have multiple cameras in one scene
  - Either with only one active at a time OR all active and rendering to different parts of the player’s screen
  - Good for splitscreen multiplayer, rendering minimaps, and overlaying weapons in first person view

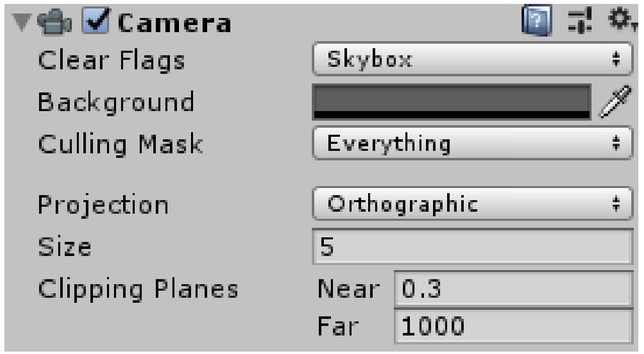
Camera Properties

- https://docs.unity3d.com/Manual/class-Camera.html
- Clear Flags – Every frame the camera view is re-rendered, and this property specifies what should be rendered first.
  - Skybox – a detailed image that is wrapped around the inside of a box (or sphere). The camera is on the inside of this box and the box edges are infinity far away, giving the illusion of a sky and horizon.
  - Solid color – Just sets the a uniform color over the entire camera. Great if there is no sky or for 2D games where the background will be a flat image itself within the scene.

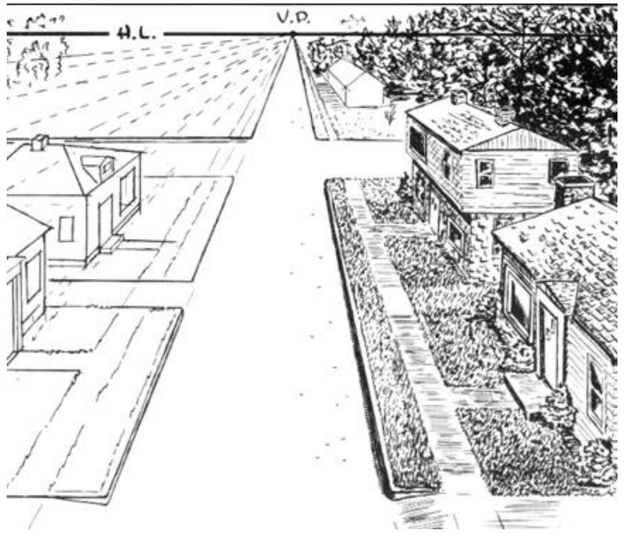
- Clipping planes
  - Near plane – anything closer to the camera than this distance will not be rendered
  - Far plane – anything further than this plane will not be rendered

Camera Projection

- **Projection Matrix**
  - Determines how each point in 3D space is transformed to be rendered to a 2D flat screen (e.g. 3D points are “projected” onto a 2D space)

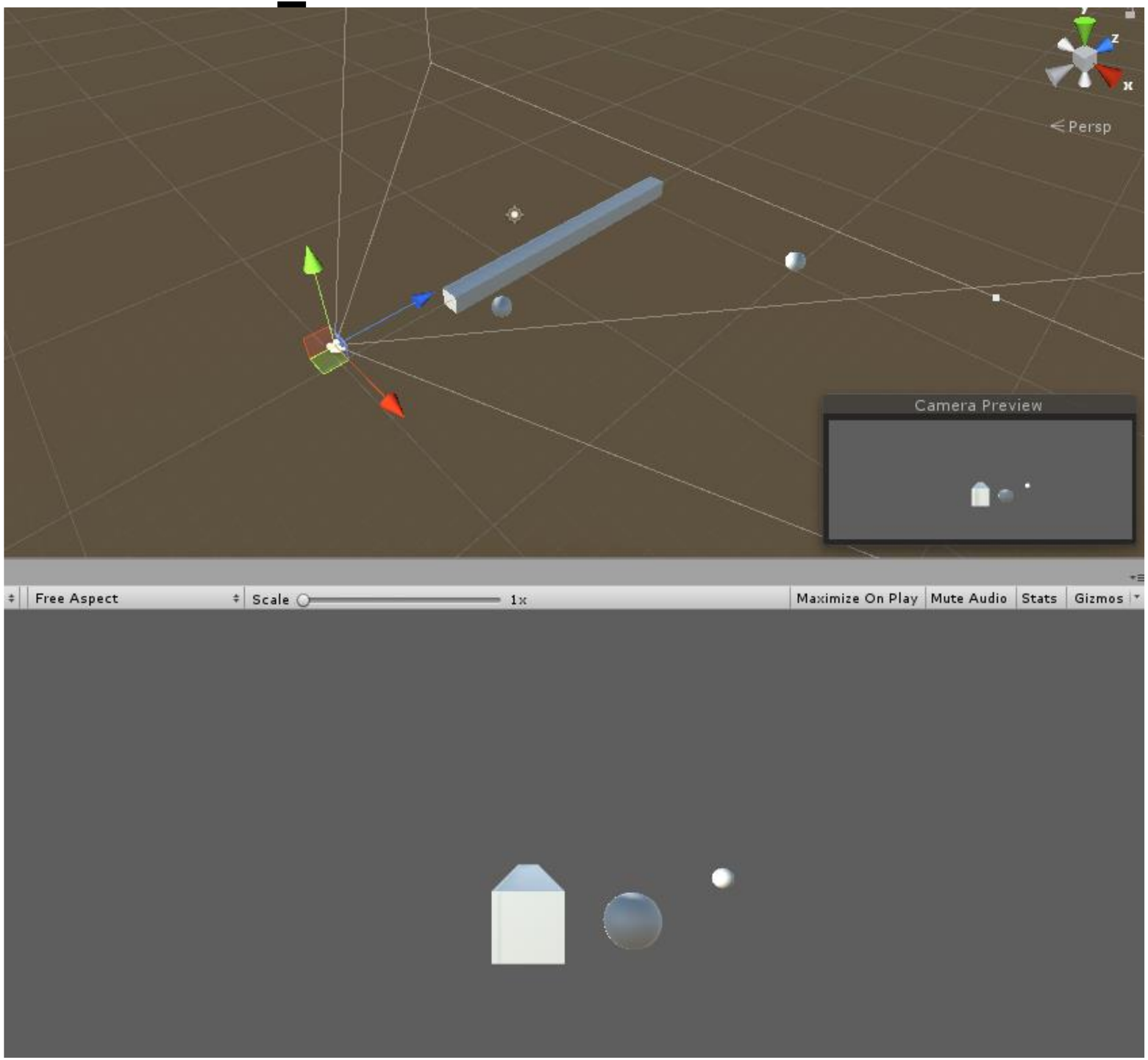


- **Perspective**
  - All 3 dimensions are seen
  - How human eyes see the world
  - View gets wider as it gets further away from the eyes/camera
  - As parallel lines get further from our eyes, we perceive them to converge at the horizon
  - This results in objects further away appearing smaller
  - Allows us to see sides of an object slightly unaligned to our eyes

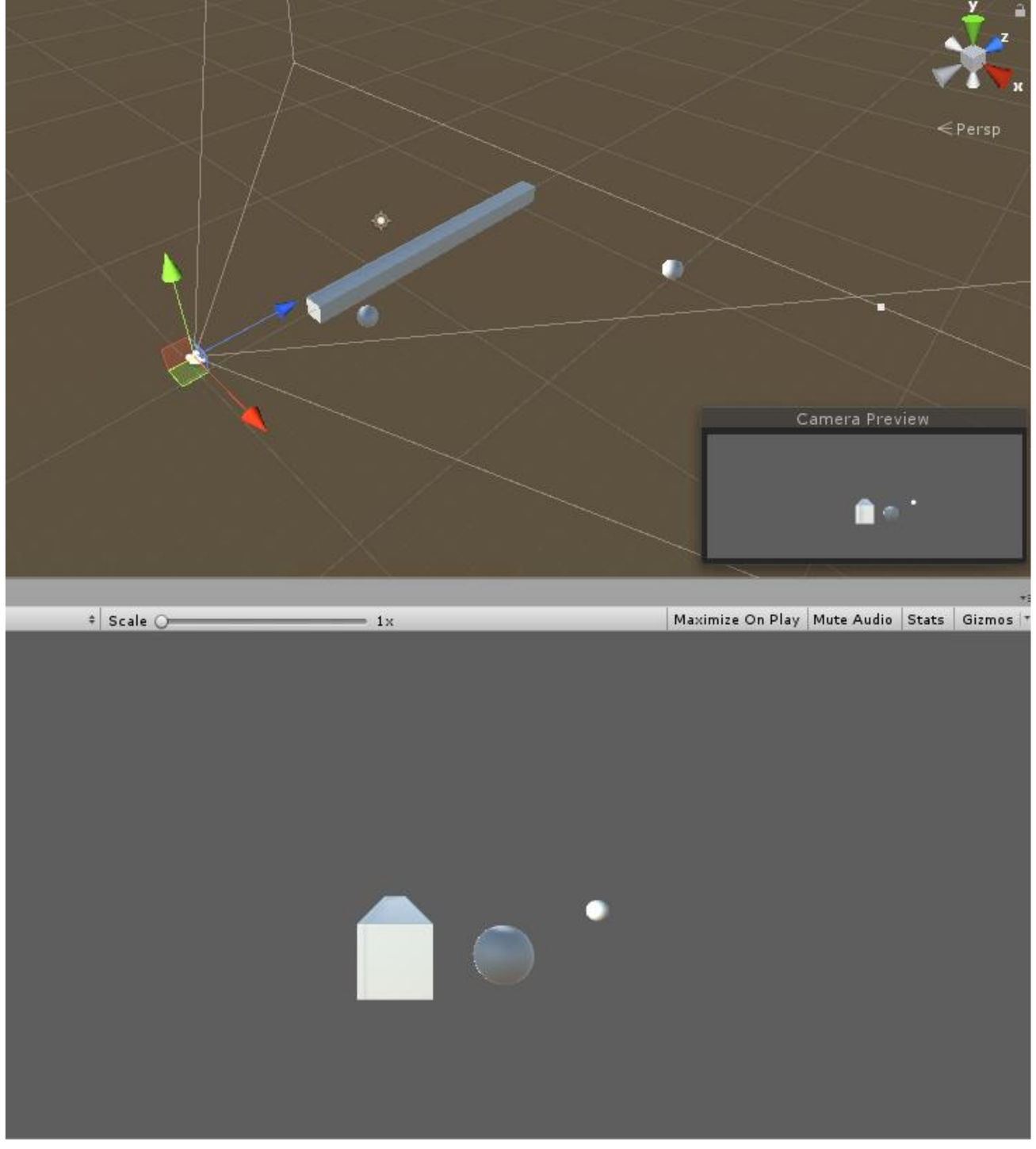


- **Orthographic**
  - Only 2 dimensions are seen
  - View stays the same width as it gets further from the eyes/camera
  - Parallel lines do not appear to converge
  - Therefore, distance to the camera is irrelevant (there is no horizon!)

Orthographic View



Perspective View



Lighting

- In the physical world, without any lights (e.g. the sun, moon, fire, artificial light sources) everything is black
  - Our eyes don’t see objects – they see the light bouncing off of objects!

- Game camera work the same way
  - A scene needs light sources for anything to be seen

- Multiple different types of lights
  - Each with their own point of origin and how the light rays travel out from that point
- Also with:
  - Color - reacts with object material color to give a final overall color that is seen
  - Intensity – how powerful the light is – e.g. the wattage/lumens of a light bulb

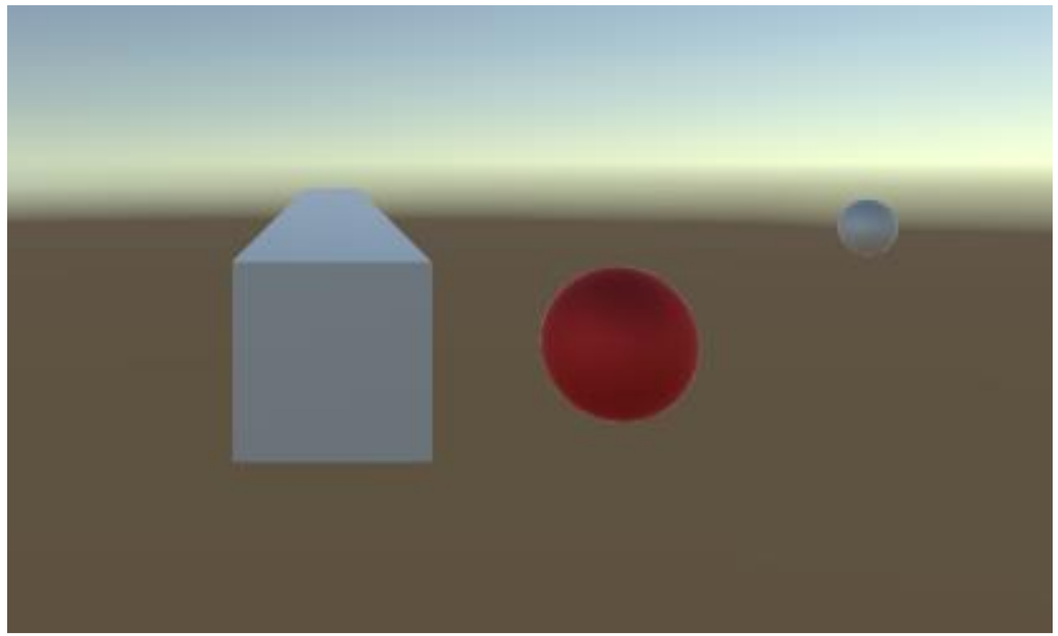
https://docs.unity3d.com/Manual/Lighting.html

Ambient Lights

- An ever present light glow to minimally allow object to be seen without shadows being cast
  - Imagine a room with perfect lighting where you don’t know where the light is coming from but you can still see everything
  - Mostly just shows basic background color and foreground color

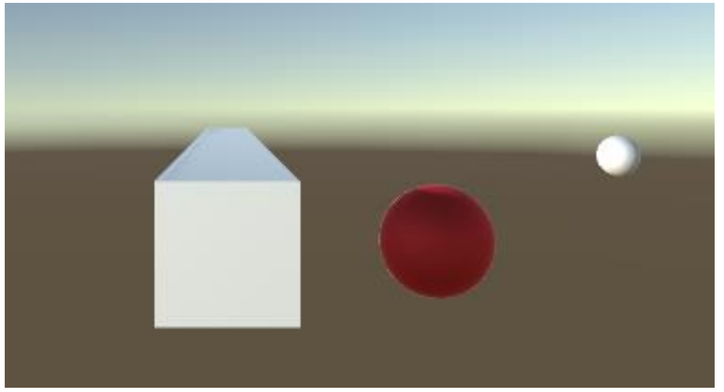
Environment Light

- Comes from a skybox, simulating the sun or moon in the skybox
- Changed through Window->Rendering->Lighting Settings->Environment Lighting



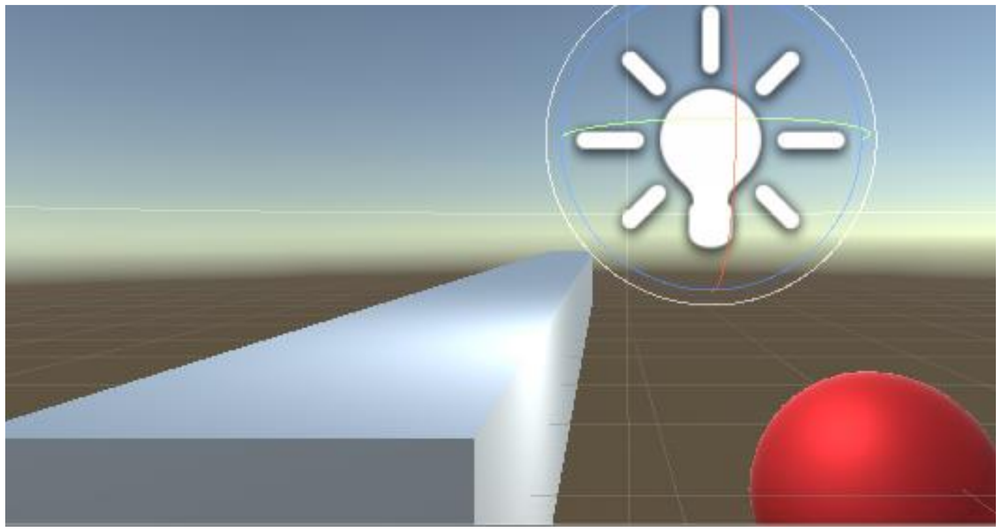
Directional Light

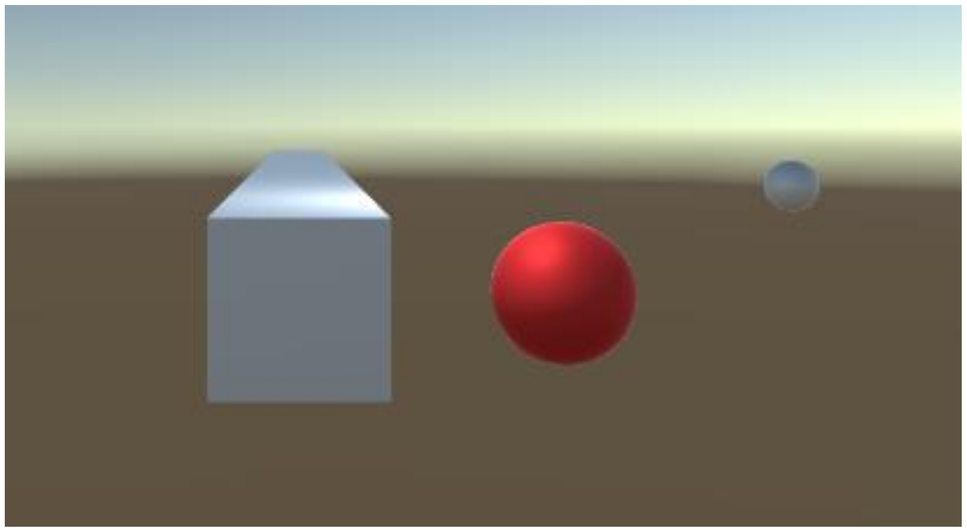
- Similar to ambient light but there can be many of them and it casts shadows
- A light with only a direction, no position
  - Light source is infinitely far away
  - Light rays travel in straight lines
- Good for simulating sun or just generally brightening a scene from one side



Point Light

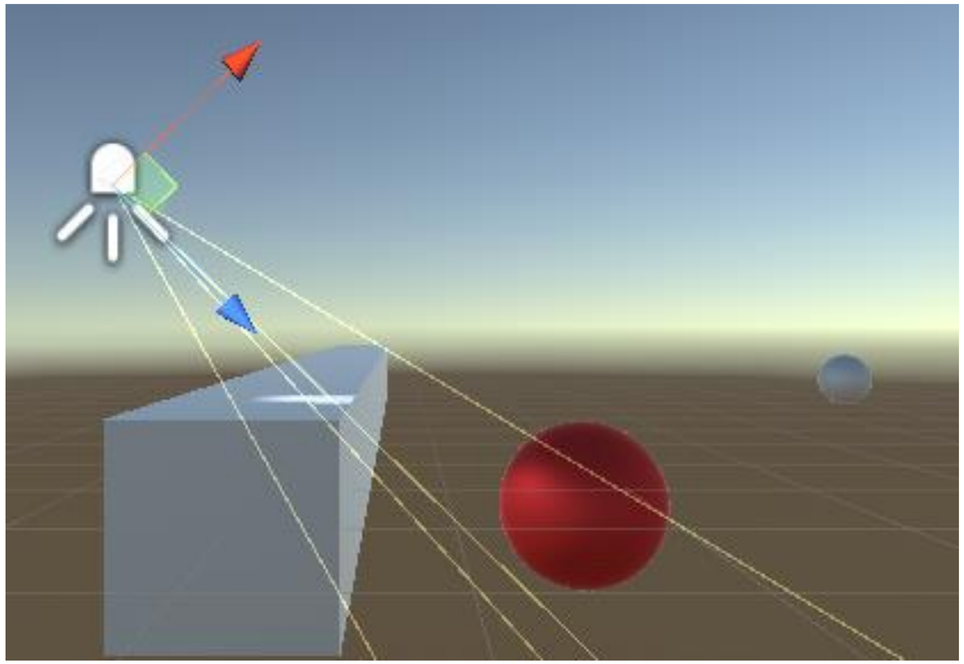
- Light emits from a single point
- Emits in a sphere in all directions around that point.
- Light gets dimmer the further it is from the source
- Good for simulating unconstrained light sources, like an explosion

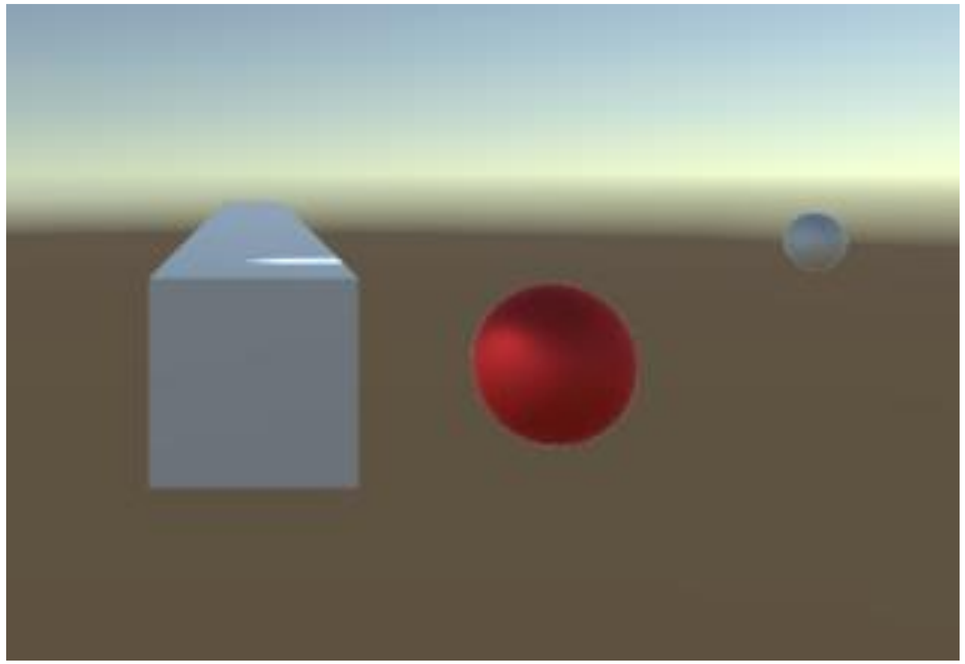




Spot Light

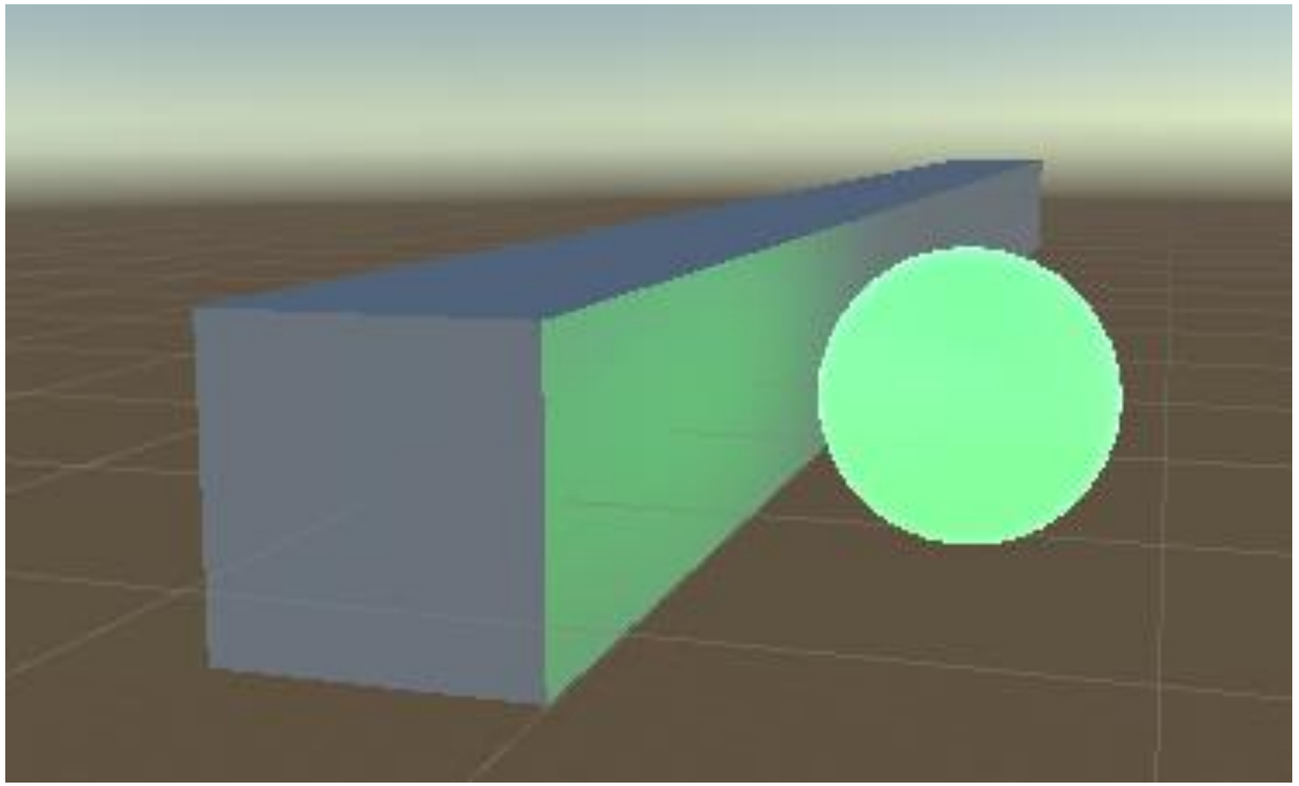
- Light emits from a single point
- But comes out in a cone away from the light source
- Light gets dimmer and disperses the further it is from the source
- Good for constrained lights such as ceiling lights and lamps





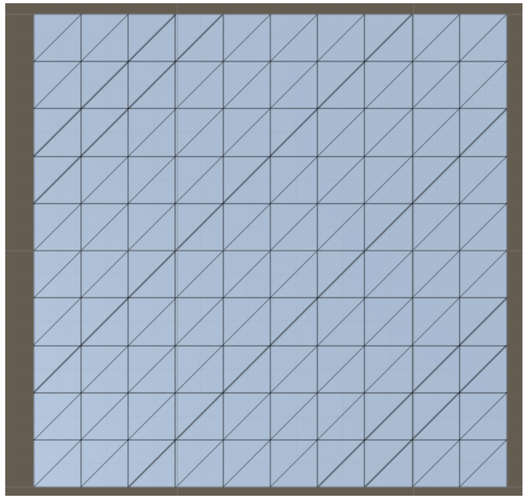
Emissive

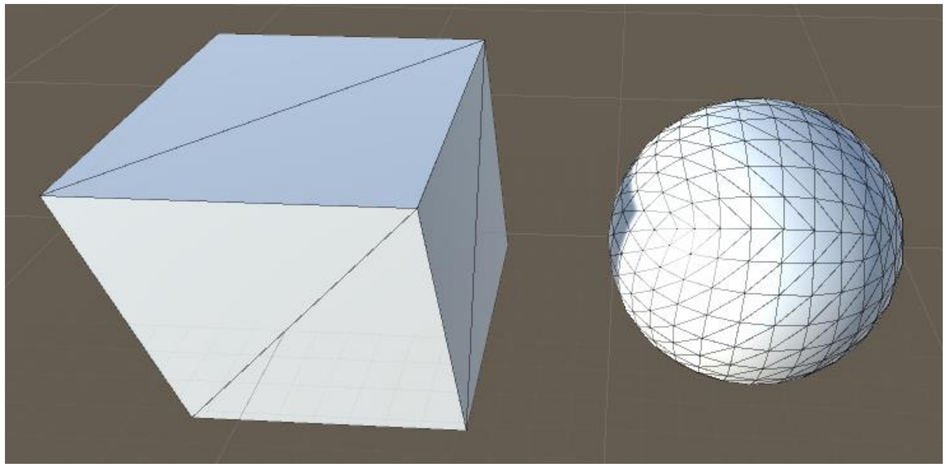
- Light is emitted from the color properties of an objects material (see later slides)
- Good for glowing object (e.g. neon signs)



3D Models

- A 3D Model is made up of:
  - Vertices – points in 3D space.
  - Edges – lines that join vertices together
  - Triangles – 3 vertices joined together with edges that can be rendered as a flat shape
  - Surfaces – multiple triangles joined together to form the outside face of a 3D shape.



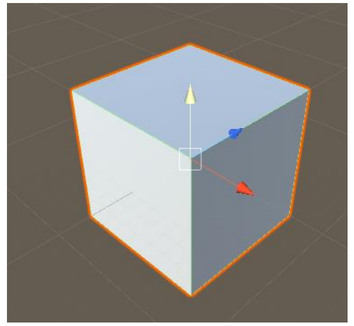


- <u>Unity is not a 3D modelling tool!</u>
  - You will need to create/edit complex meshes with an external 3D tool such as Maya, Blender, 3ds Max, Zbrush etc.

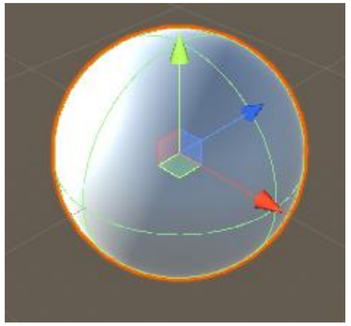
3D Primitives

- 3D Primitives = basic geometry
  - Can be combined to make more complex geometry
  - Good for rapid prototyping and basic objects (e.g. walls, floors, doors, video screens, etc.)

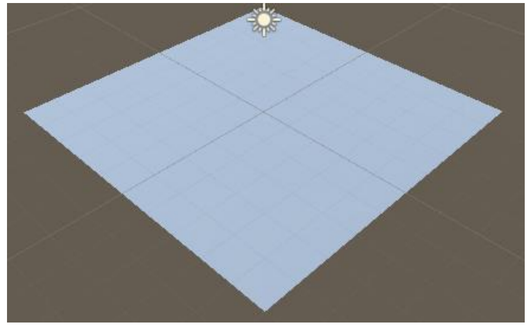
- Unity has 6 primitives
  - Defined at https://docs.unity3d.com/Manual/PrimitiveObjects.html
  - Other 3D tools often also have cones, pyramids, and torus rings



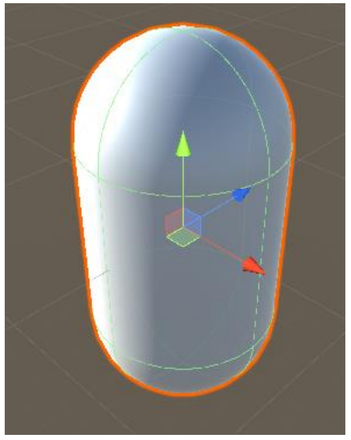
Cube



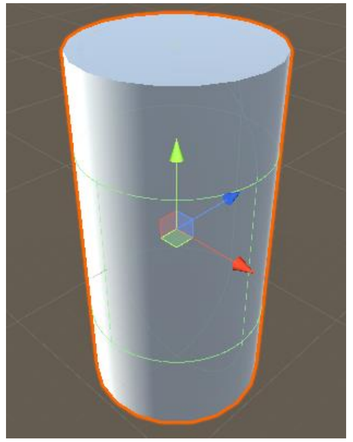
Sphere



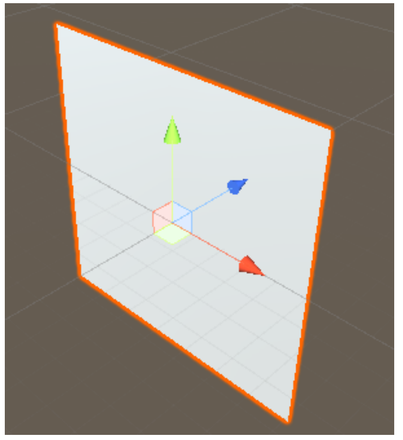
Plane (121 Vertices)



Capsule



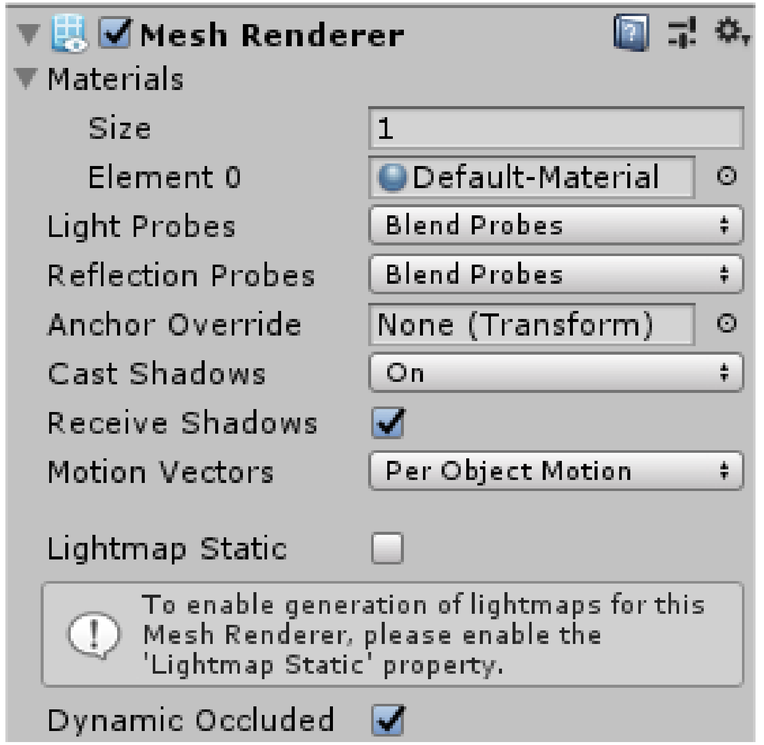
Cylinder



Quad (4 Vertices)

Renderer Components

- In-built Unity component (script) for showing (rendering) 2D images and 3D objects to the scene
  - Tells the graphics card how to handle the image/object and how to display it on the users screen
- MeshRenderer
  - Used for 3D objects
  - Determines what Materials are on the object
  - How shadows and reflections interacts with the object
  - Whether the object is still rendered if it isn’t being seen by a camera



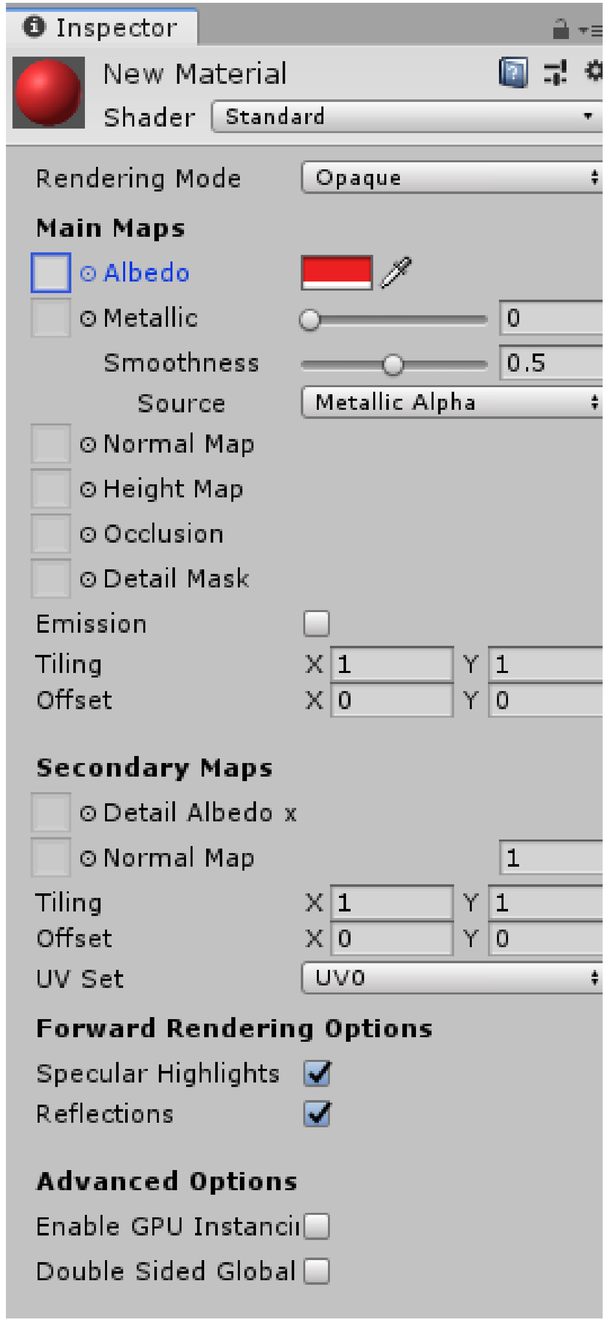
Material

- Meshes only specify the surface **shape** of an object

- Materials specify the **detail** on the surface of the object
  - Colour
  - Texture (e.g. patterns on the surface)
  - How light bounces off the objects

- Sperate assets in the Project Window
  - Attached to Renderer components on game objects in the scene

- Shader – code that dictates how a material is applied to a mesh
  - And additional effects once it has been applied



Materials Continued

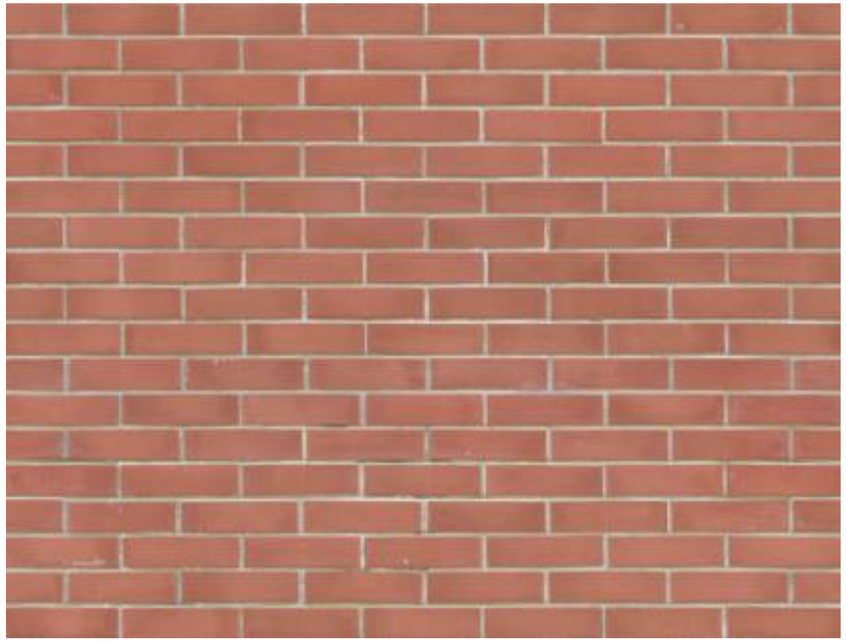
- Albedo – Overall colour tint and transparency of the material
  - Values are between 0 and 256 for (r,g,b,a) – (red, green, blue, alpha)
  - Same as colour values for e.g. HTML
  - Anything over 256 is constrained to 256
  - https://docs.unity3d.com/Manual/StandardShaderMaterialParameterAlbedoColor.html

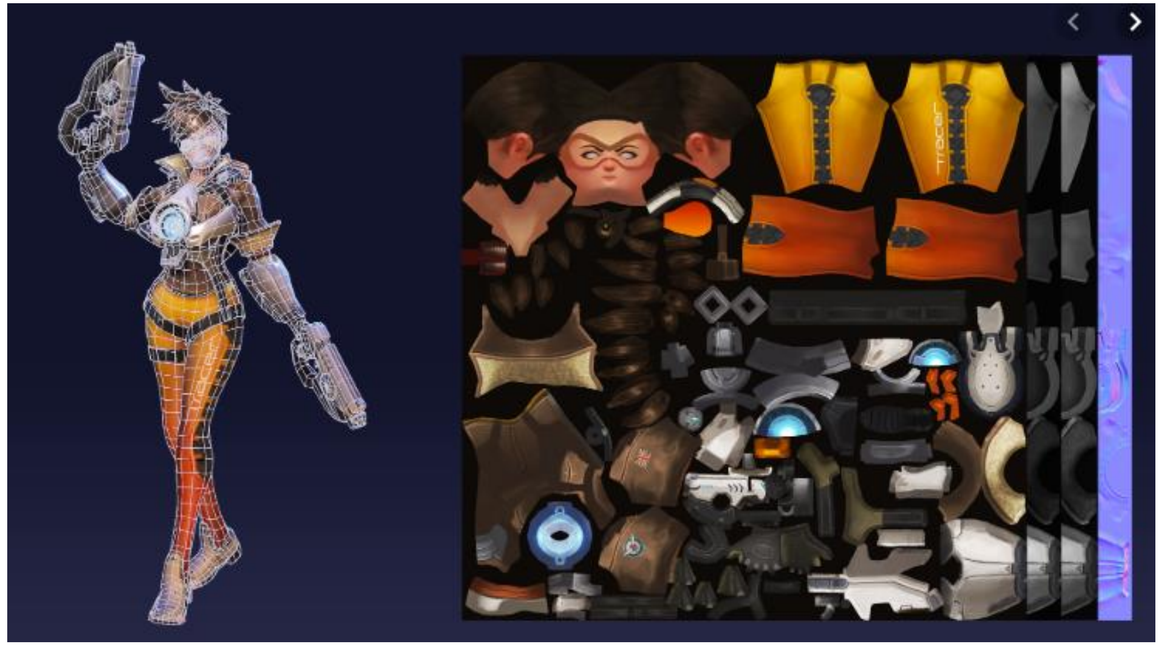
- Metallic and Smoothness – how light distributes over the surface of the material
  - Does light uniformly spread over the material?
  - Or does it create a single bright point that bleeds off quickly to darker areas?
  - https://docs.unity3d.com/Manual/StandardShaderMaterialParameterMetallic.html

Textures

- 2D images that are used in a material to be wrapped around a 3D mesh

- UV Map – creates points (u,v) on the 2D texture to align with the vertices of a 3D Mesh (x,y,z)
  - Allows the texture to be stretched over the mesh in an appropriate way



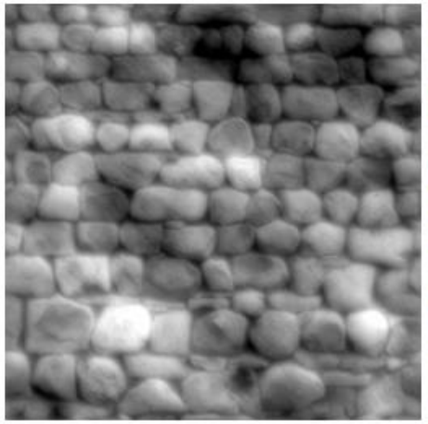


Materials Continued

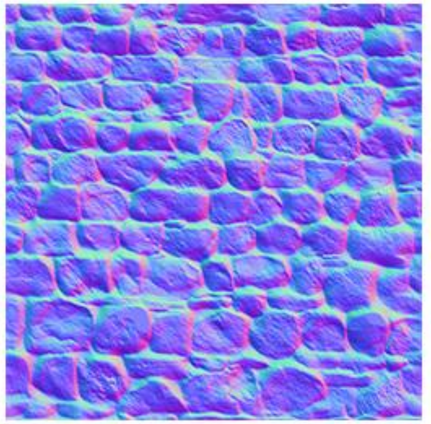
- Normal Maps – specifies which direction each (u,v) point is facing to determine how light will bounce off the 2D image
  - “Bump mapping” – Normal maps allow for bumps and grooves to be added and react naturally to lighting
  - Made more intense and realistic through Height Maps
  - https://docs.unity3d.com/Manual/StandardShaderMaterialParameterNormalMap.html

Without bump mapping

With bump mapping



Height Map



Normal Map

Sprites

- 2D is MUCH simpler
  - No mesh
  - No material
  - No light interaction
  - No albedo, UVs, normals, etc.
  - Just a sprite

- Sprite == 2D image
  - Displayed in game to represent a single game object

- Sprite sheet – a single image file with multiple sprites that is cut up at runtime to create multiple objects or multiple versions of the same object (e.g. in sprite animation)
  - Improves on memory and computation performance, as well as project management

Particle Effects

- The simulation of lights particles (i.e. small sprites or meshes) that
  - Move in a specific pattern
  - Are affected by physics
  - Collide with other objects
  - Receive and emit light

- Used to represent anything small or made up of small pieces
  - Sand / dust
  - Fire
  - Smoke
  - Explosions
  - Magic spells
  - Etc.

Shuriken Particle System

- Unity has a complex and very powerful particle engine, known as Shuriken

- Allow you to adjust every little detail of how the particles look, are created, move, lifetime, light emission, noise, etc.
  - All without any code!
  - But, because it is a component, you can also edit almost all of it through code to make your particle system dynamically react to gameplay.

- We won’t go into detail on particle systems in this subject
  - Just know that they are there, how to use them in your game, and how to make basic edits to them
  - People have careers in just making awesome particle effect for films and games