CE318/CE818 High-Level Games Development

Models and Physics – Lecture 3

Dr Aikaterini (Katerina) Bourazeri

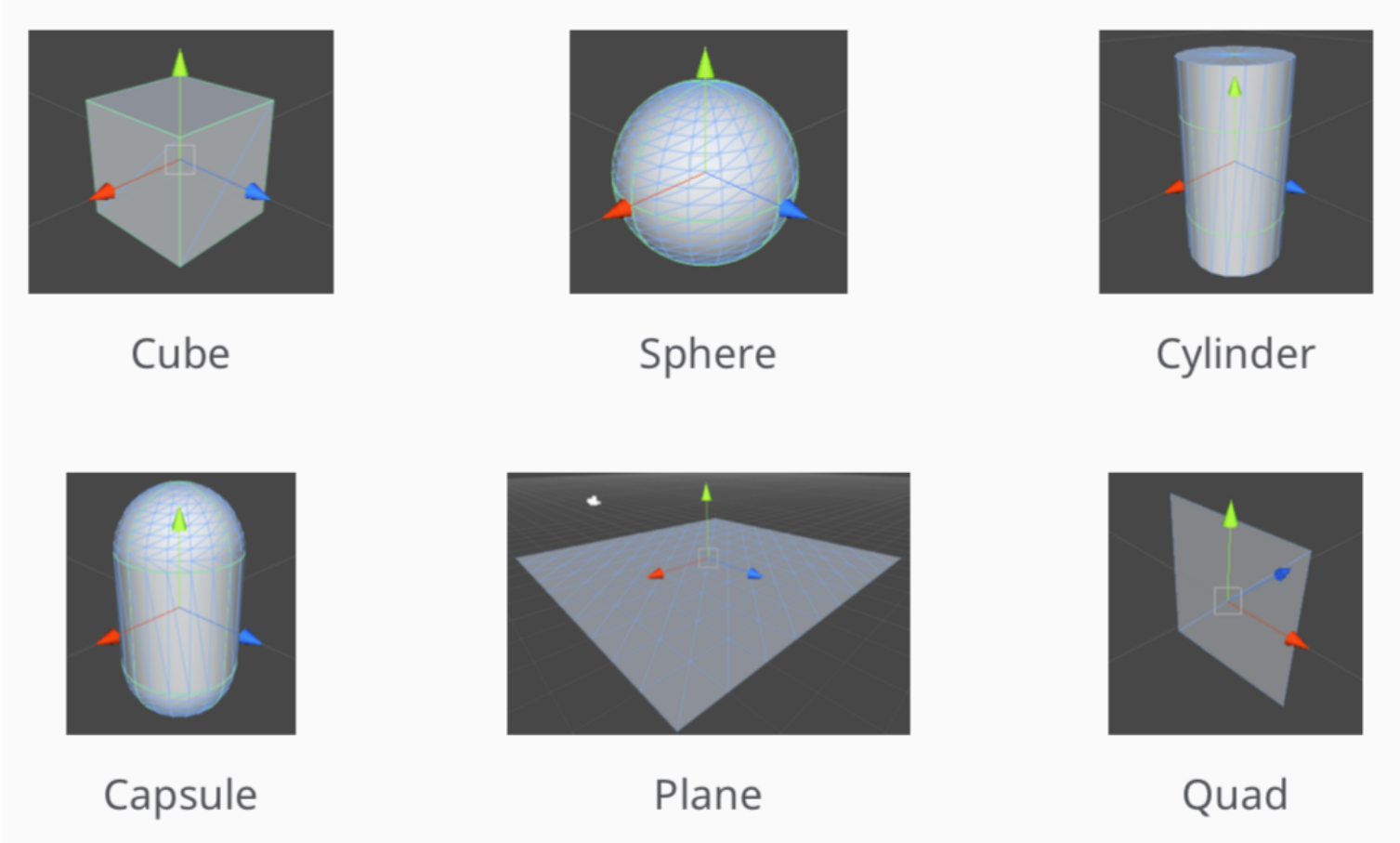
30^{th} October 2023

Introduction

- In today's lecture, we will talk about:
  - How our objects *look* – models
  - How our objects *act* – physics
- *Neither* of these are things you want to hand-code.
- One of the main benefits of using game engines is that you don't have to.

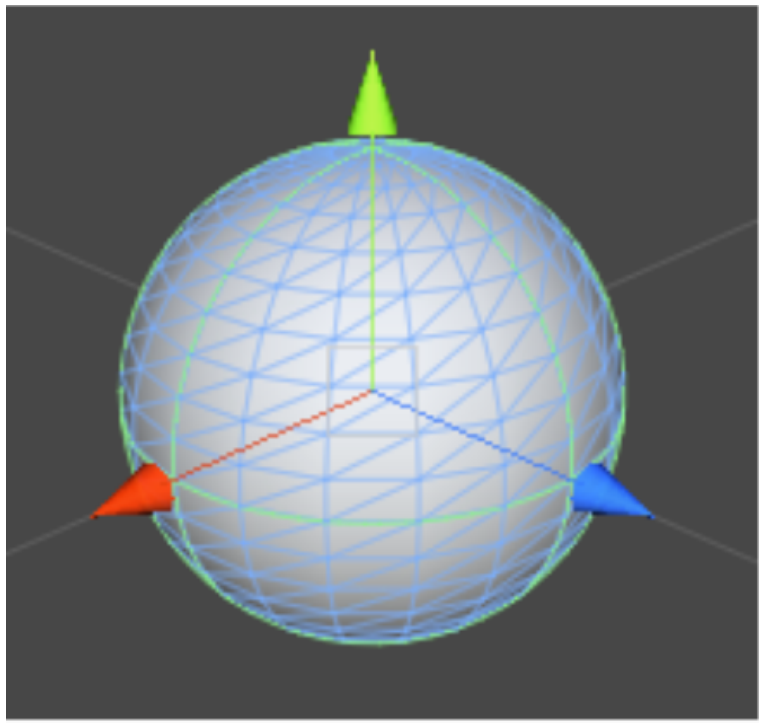
Basic Meshes

Unity has some pre-defined meshes which can be used for your games.



Vertices, Triangles and Meshes

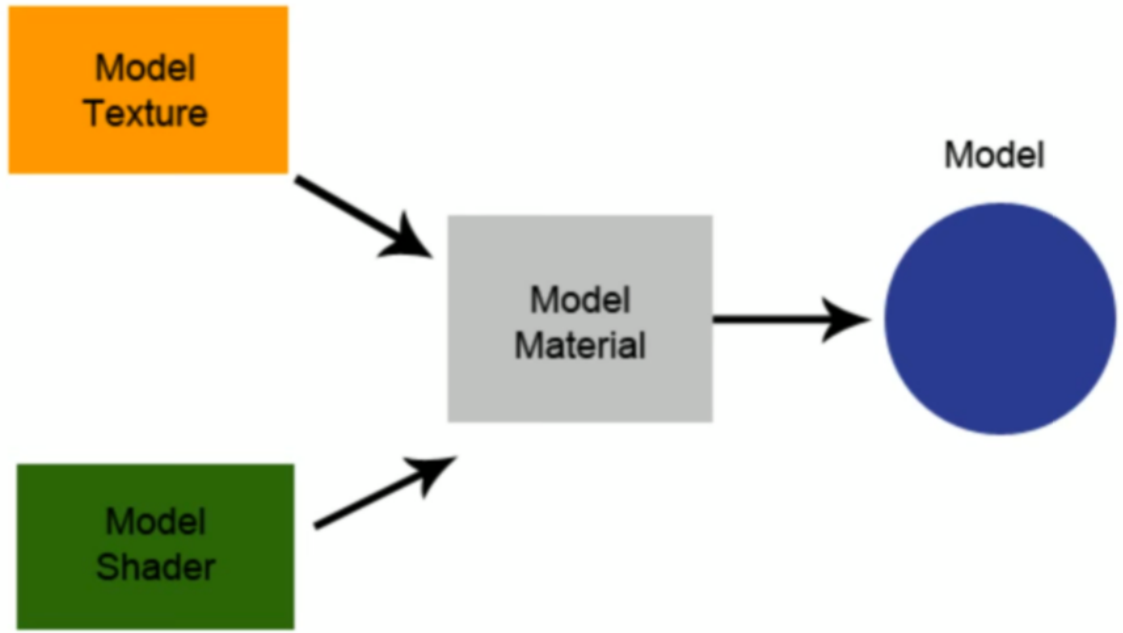
- A mesh is a collection of 3D points (vertices).
- Vertices are combined to form triangles.
- We use triangles because:
  - 3 points form a unique triangle with sides that don't cross each other.
  - The three points determine a triangle in a unique plane.



3D Models

- However, for our games we will often need more complex meshes.
- To do this we can use *models*.
- There are two types of formats:
  - Exported 3D formats: FBX, OBJ
  - Proprietary 3D formats: .Max, .Blend
- Unity can read FBX, dae, 3DS, dxf, obj files.

Using 3D Models

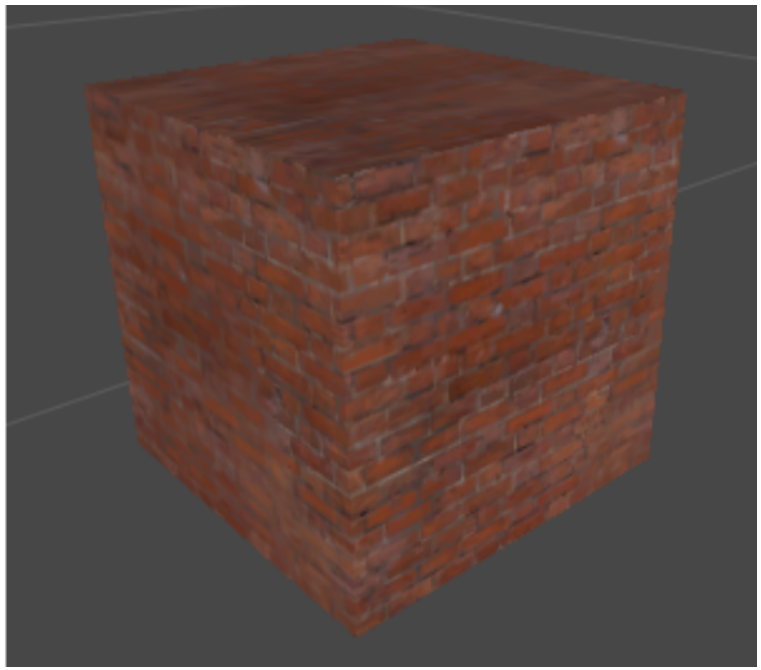


- A model has a Model Material applied to it.
- A material can be seen as the skin of the mesh.
- Every mesh needs a material to be seen on the screen.
- The model material has two parts: texture and shader.

Materials

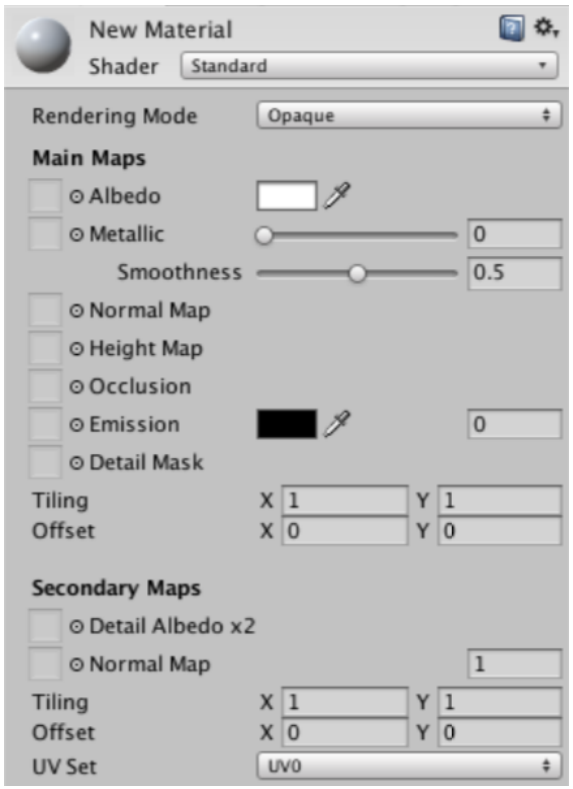
Example

- Unity provides a basic material for every mesh – you have already seen it.
- One of its basic parameters is a texture.
- You can create new materials and assign textures to them.



Shaders

- Shaders in Unity are used through Materials. The shader controls the properties of objects in the scene.
- The default shader is the Standard Shader, used in the default material.
- It is very versatile and can be configured in multiple ways.



The Standard Shader I

- **Rendering Mode:** Opaque (default).
  - Opaque (default).
  - Cutout: the alpha channel of the diffuse image is used to cut out parts of the texture.
  - Fade: to make the object invisible.
  - Transparent: to make it transparent, preserving its reflectivity.

- **Main Maps:** Properties defined by texture maps.
  - Albedo: combination of an optional texture and a colour value to tint the texture (white: unaffected).
  - Metallic: "metalness" of the material's surface, defined by a texture or a slider. The texture colour defines how metallic each point is (0 red for totally metallic, 255 red for no metallic).
  - Normal map: defines apparent bumpiness of the surface.

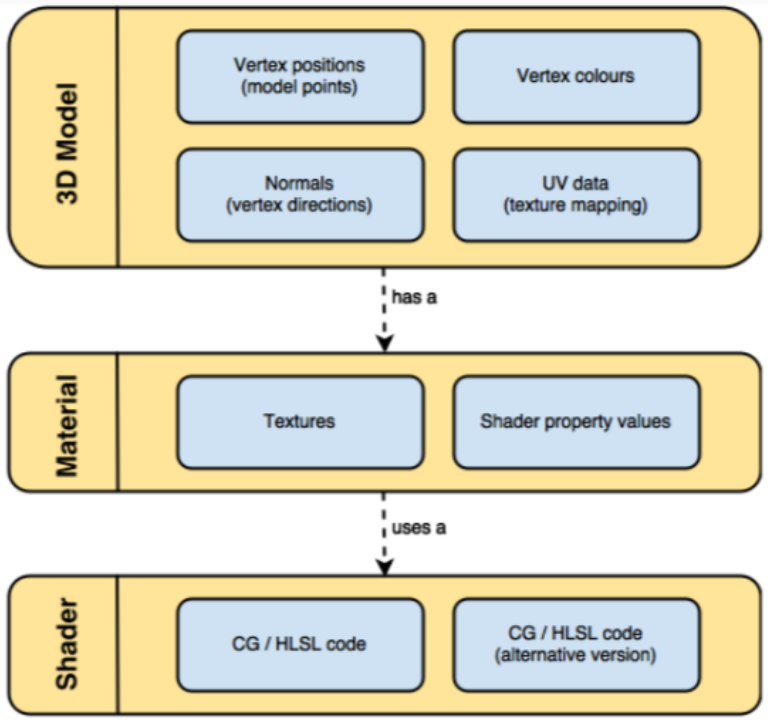
The Standard Shader II

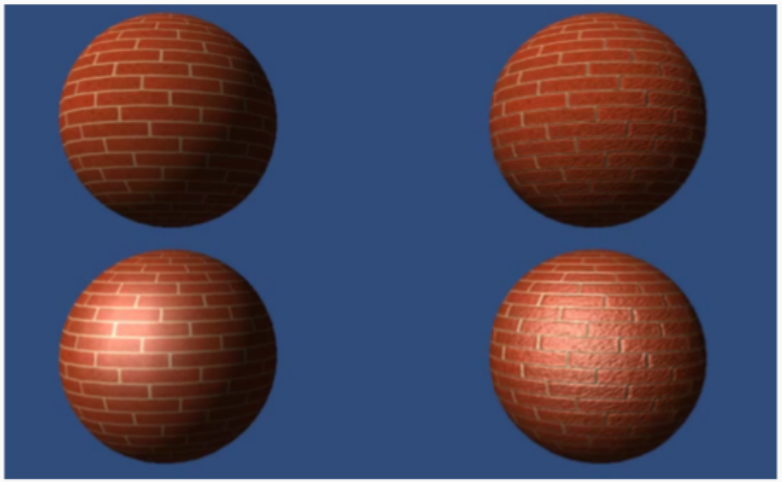
* Height map: defines apparent height of the surface.
* Occlusion: how it reacts to ambient light.
* Emission: to make the material contribute to the scene lighting.
* Tiling and offset control position of the map.

- **Secondary Maps:** Define additional surface details, drawn on top of the main maps.

The Standard Shader has an alternative (Standard, Specular Setup) that uses *Specular* instead of *Metallic*. It can be also defined with a texture and colour, and it is analogous to the metallic property in the Standard shader.

The Whole Picture





3D Physics

- Rigidbodies
- 3D Colliders: Box, Capsule, Mesh, Sphere, Wheel, Terrain
- Joints: Hinge, Spring, Character, Configurable, Fixed
- Forces and Torque

RigidBody

- A RigidBody is the main component that enables physical behaviour for an object.
- With a RigidBody attached, the object will immediately respond to gravity.
- The RigidBody component will take over movement of the object it is attached to – so you shouldn't edit the transform directly (rotation and position).
- Instead you should apply forces to push the job and let the physics engine calculate the results.
- You can mark a RigidBody has kinematic in order to remove it from the control of the physics engine and allow it to be moved kinematically from a script.

Sleeping

- It is expensive to compute physics for every object in every scene in every frame.
- So, in order to prevent updates to stationary (or near stationary) objects – Physics engines put objects to sleep.
- Unity's Physics engine(s) are no exception.
- If an object moves at less than a certain speed, the engine will assume it has come to a halt and put it to sleep.
- When another force or collision occurs it will wake up.

Colliders & Triggers in Unity3D

Many games simulate real-world physics, and collisions are a crucial part of these simulations. When we develop games, most of the time we start or stop events based on other events. We use colliders and/or triggers for these purposes:

- Colliders are components that allow the physics engine to handle the collisions.
- We can determine which objects collide with each other, or how objects behave under collisions.
- Triggers are special setups of colliders; trigger events when more than one objects overlap.
- If an object has a collider configured as a trigger, that object does not collide with any other object but overlaps instead.

Combining Colliders

- You can create more complex colliders by attaching multiple colliders to the object and offsetting them.
- You can also add additional colliders to child objects, but there should only be one RigidBody (attached to the root object).
- A good general rule is to use Mesh colliders for scenery and use combined primitive colliders for objects that will move.

Static Colliders

- A game object with a collider but no RigidBody.
- Usage: level geometry.
  - The physics engine assumes that static colliders never move or change.
  - As a result, don't disable/enable, scale or move static colliders during gameplay.

RigidBody Collider

- A GameObject with a collider and a non-kinematic RigidBody.
- Usage: basically everything.
  - Fully simulated physics engine and can react to forces from scripts.
  - They can collide with other objects (including static colliders).

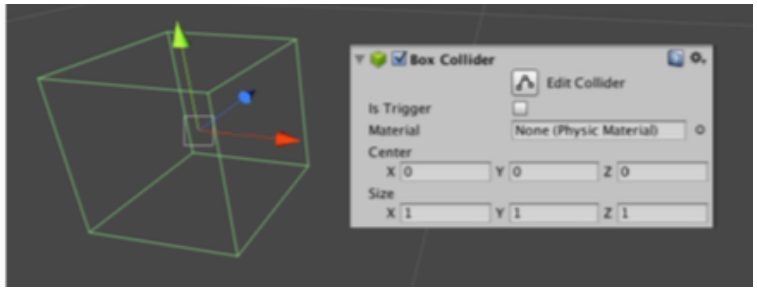
Kinematic RigidBody Collider

- A GameObject with a collider and a kinematic RigidBody.
- Usage: Static colliders that can move.
  - You can modify the transform directly but the object won't respond to forces.
  - These are basically used for objects that could be enabled/disabled or moved occasionally.
  - Example: A sliding door that should normally act as an immovable physical obstacle but can be opened when necessary.

Box Collider

The Box Collider is a basic cube-shaped collision primitive. Box colliders are obviously useful for anything roughy box-shaped, such as a crate or a chest. However, a thin box can be used as a floor, wall or ramp and the box shape is also a useful element in a compound collider.

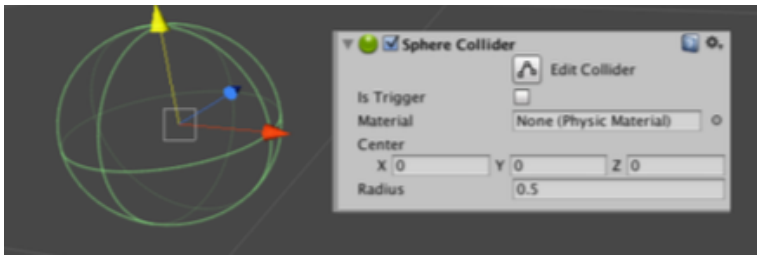
- Material: Reference to the Physics Material that determines how this Collider interacts with others.
- Center: The position of the Collider in the object's local space.
- Size: The size of the Collider in the X, Y, Z directions.



Sphere Collider

The Sphere Collider is a basic sphere-shaped collision primitive. The collider can be resized via the Radius property but cannot be scaled along the three axes independently. As well as the obvious use for spherical objects like tennis balls, etc., the sphere also works well for falling boulders and other objects that need to roll and tumble.

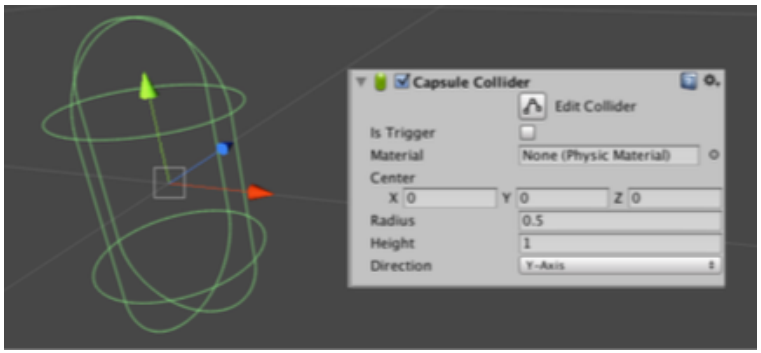
- Material: Reference to the Physics Material that determines how this Collider interacts with others.
- Center: The position of the Collider in the object's local space.
- Radius: The radius of the sphere.



Capsule Collider

The Capsule Collider is made of two half-spheres joined together by a cylinder. It is the same shape as the Capsule primitive.

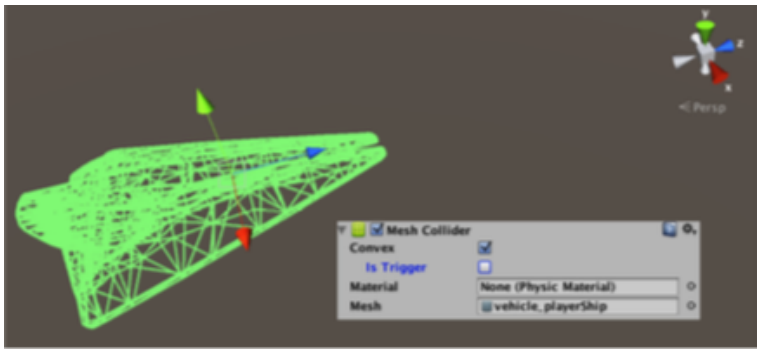
- Material: Reference to the Physics Material that determines how this Collider interacts with others.
- Center: The position of the Collider in the object's local space.
- Radius: The radius of the Collider's local width.
- Height: The total height of the Collider.
- Direction: The axis of the capsule's lengthwise orientation in the object's local space.



Mesh Collider

A collider built out of a Mesh Asset. Although it allows for more accurate collisions it is much more computationally expensive. By default they won't collide with other mesh colliders.

- Convex: Set it to *true* if the collider has no holes or entrances.
  - If enabled, this Mesh Collider will collide with other Mesh Colliders.
  - Convex Mesh Colliders are limited to 255 triangles.
  - Needs to be convex to work with a RigidBody.
- Material: Reference to the Physics Material that determines how this Collider interacts with others.
- Mesh: Reference to the Mesh to use for collisions.



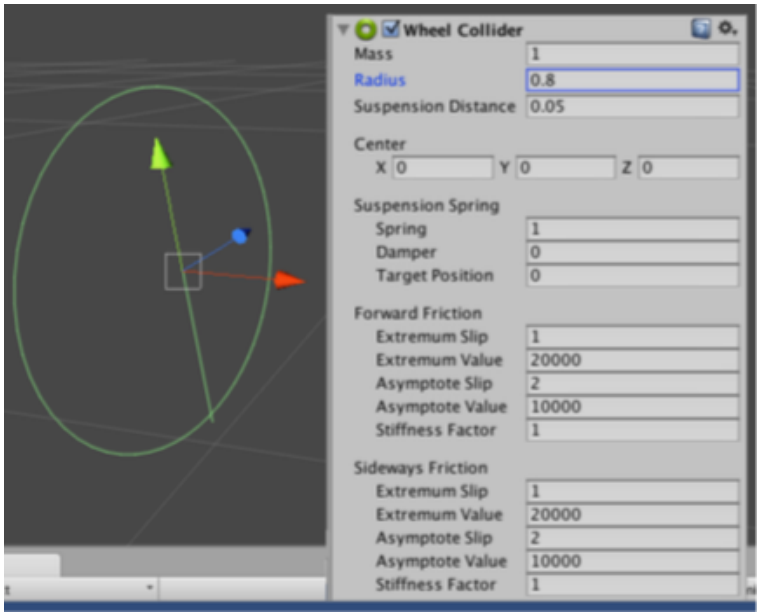
Terrain Collider

The Terrain Collider takes a Terrain and builds its Collider based on that terrain.

- Terrain Data: The terrain data.
- Create Tree Colliders: When selected Tree Colliders will be created.

Wheel Collider

The Wheel Collider is a special collider for grounded vehicles. It has built-in collision detection, wheel physics, and a slip-based tire friction model. It can be used for objects other than wheels, but it is specifically designed for vehicles with wheels.



Forces

- Forces are the recommended way to move Physics Objects.
- They are any interaction which is used to change the motion of an object.
- In Unity, they are vectors:
  - Linear – Force
  - Rotational – Torque
- They can be applied to objects or to a specific point.
- Applying forces to a RigidBody will result in a change in its transform.

Applying Forces

- Forces should be **applied** in the FixedUpdate callback.
- FixedUpdate is called immediately before each physics update.
- The update callback may not necessarily co-inside with the physics engine.
- Forces are dampened by the RigidBody's drag property.

```
//can use  vector3 or the three components
public void AddForce(Vector3 force, ForceMode mode = ForceMode.Force);

public void AddForce(float x, float y, float z, ForceMode mode = ForceMode.Force);

rigidbody.AddForce(Vector3.up * 10);
```

Torque

- Torque is force around an axis.
- *AddTorque* adds a torque force to a RigidBody.
- As a result the RigidBody will start spinning around the *torque* axis.

```
//can use  vector3 or the three components
public void AddTorque(Vector3 force, ForceMode mode = ForceMode.Force);

public void AddTorque(float x, float y, float z, ForceMode mode = ForceMode.Force);

float v = Input.GetAxis("Vertical") * amount;
rigidbody.AddTorque(transform.right * v);
```

Other Forces

- *AddRelativeForce* Adds a force to the RigidBody relative to its **local** coordinate system.
- *AddForceAtPosition* applies a *force* at a given *position*.
  - As a result this will apply a torque and force on the object.
  - For realistic effects position should be approximately in the range of the surface of the RigidBody.

Force Modes

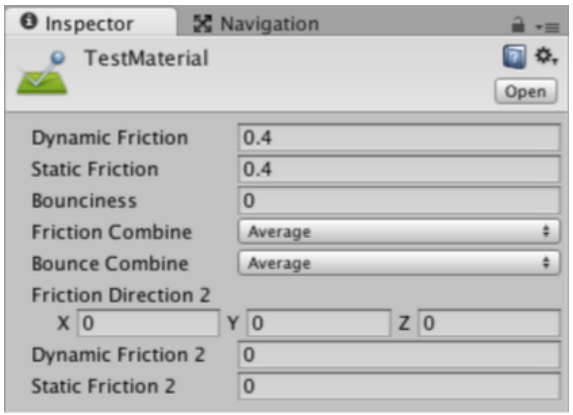
In Unity, there are a few different modes which forces can be applied in:

- **Force** – Continuous changes that are affected by mass (movement depends on the mass of the object).
- **Acceleration** – Add a continuous acceleration to the RigidBody, ignoring its mass (movement **does not** depend on the mass of the object).
- **Impulse** – Add an instant force impulse to the RigidBody, using its mass. All force is applied at once.
- **VelocityChange** – Add an instant velocity change to the RigidBody, ignoring its mass.

Physics Materials

The Physics Material is used to adjust friction and bouncing effects of colliding objects. They are assigned to the collider component of a game object.

- Dynamic Friction: The friction used when already moving. Usually a value from 0 to 1. A value of zero feels like ice, a value of 1 will make it come to rest very quickly unless a lot of force or gravity pushes the object.
- Static Friction: The friction used when an object is laying still on a surface. Usually a value from 0 to 1. A value of zero feels like ice, a value of 1 will make it very hard to get the object moving.

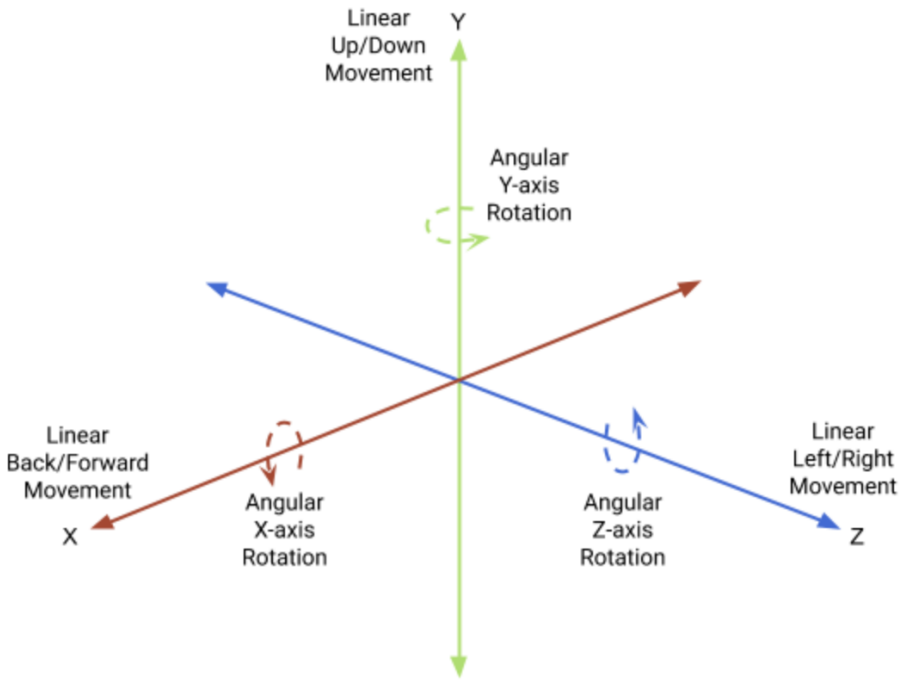


Physics Materials

- Bounciness: How bouncy is the surface? A value of 0 will not bounce. A value of 1 will bounce without any loss of energy.
- Friction Combine: How the friction of two colliding objects is combined.
  - **Average** – The two friction values are averaged.
  - **Minimum** – The smallest of the two values is used.
  - **Maximum** – The largest of the two values is used.
  - **Multiply** – The friction values are multiplied with each other.
- Bounce Combine: How the bounciness of two colliding objects is combined. It has the same modes as Friction Combine.
- Friction Direction 2: Specific friction for a particular direction (enabled only if different than 0).
- Dynamic Friction 2: Dynamic friction along *Friction Direction* 2.
- Static Friction 2: Static friction along *Friction Direction* 2.

Joints

Joints apply forces that move RigidBodies, and joint limits restrict that movement. We want a joint to allow at least some freedom of motion.



Joints Types

* **Fixed Joint:** sticks objects together permanently or temporarily. **Example:** *"sticky grenade"* – as the enemy moves around, the joint will keep the grenade stuck to them.
* **Spring Joint:** acts like a piece of elastic that tries to pull the two anchor points together to the exact same position. **Example:** connect character's body parts together, allowing them to flex to and from each other.
* **Hinge Joint:** rotates at the point specified by an Anchor property, moving around a specified Axis property. **Example:** hinge of a *door*.
* **Character Joint:** is similar to a human joint, it is also called ball and socket joint. **Example:** hip or shoulder.
* **Configurable Joint:** provides greater control of character movement; customises the movement of a ragdoll and enforces certain poses. **Example:** emulates any *skeletal joint*.

Raycasting

A **raycast** is a procedure that consists of casting a ray against all or certain colliders in the scene. Unity provides the static function *Raycast* in the class *Physics* (there are four variants):

```
public static bool Raycast(
Ray ray, //starting point and direction
RaycastHit hitInfo, //what we hit
float distance = Mathf.Infinity, //how far we can go
int layerMask = DefaultRaycastLayers //what layers we care about
);
```

Physics.Raycast returns True if the ray hits a collider, else False. If it hits something *HitInfo* will also be populated.

Raycasting – RaycastHit

RaycastHit is a struct used to get information back from a *raycast*.
Among its many fields, you can find:

- Collider collider: The collider that was hit.
- Float distance: The distance from the ray's origin to the impact point.
- Vector3 normal: The normal of the surface the ray hit.
- Vector3 point: The point in world space where the collider was hit.
- RigidBody rigidbody: The RigidBody of the collider that was hit. If the collider is not attached to a rigidbody then it is null.
- Transform transform: The Transform of the rigidbody or collider hit.

RayCast – Layer Masks

LayerMasks: By defining a layer mask, you can define colliders in specific layers that will only be affected by the raycast. Layer masks defined in your project can be accessed by (from the class LayerMask):

public static int GetMask(**params string**[] layerNames);

Example:
LayerMask collMask = LayerMask.GetMask(**"UserLayerA"**, **"UserLayerB"**);

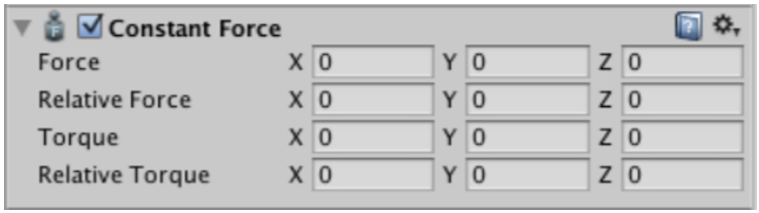
Raycasting with the supplied mask will only hit colliders assigned to layers "UserLayerA" and "UserLayerB".

Raycasting – Coding Demo

```
RaycastHit shootHit;
float range = 100f;
LayerMask shootable  = LayerMask.GetMask("Shootable");

Ray shootRay;
shootRay.origin = transform.position;
shootRay.direction = transform.forward;

//perform the raycast against gameobjects on the shootable layer:
if(Physics.Raycast (shootRay, out shootHit, range, shootableMask)) {

//try and find an EnemyHealth script on the gameobject hit
EnemyHealth = enemyHealth;
enemyHealth = shootHit.collider.GetComponent <EnemyHealth> ();

//if EnemyHealth component exists they should take damage
if(EnemyHealth !=  null) {
  enemyHealth.TakeDamage(damagePerShot, shootHit.point);
  }

}
```

Constant Force Component

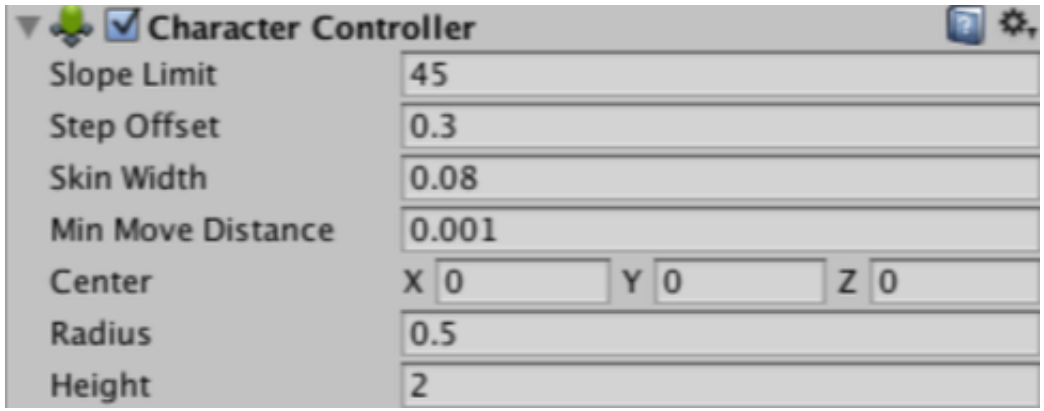
ConstantForce: Constant Force is a component that can be added to any GameObject with a RigidBody attached. While *AddForce* applies a force to the RigidBody only for one frame, thus you have to keep calling the function. *ConstantForce* on the other hand will apply the force every frame until you change the force or torque to a new value.



- Force: The vector of a force to be applied in world space.
- Relative Force: The vector of a force to be applied in the object's local space.
- Torque: The vector of a torque, in world space. The object will begin spinning around this vector. The longer the vector, the faster the rotation.

Character Controller

Character Controller is a special component available for GameObjects. RigidBodies provide **reliable** physics for your objects ...but sometimes you don't want that. Imagine the typical FPS where the characters move at high speed and jump impossible distances.



An object with a Character Controller component:

- Does not react to forces (actually, don't use it together with a RigidBody).
- Does not apply forces to other RigidBodies.
- Includes *automatically* a Capsule Collider, hence it reacts to collisions.

Character Controller Scripting

At a Scripting level, the Character Controller component includes some useful functions/variables/messages:

- `isGrounded`: Indicates if the controller is touching the ground on this frame.
- `Velocity`: Current velocity of the controller.
- `CharacterController.OnControllerColliderHit`: called when colliding with another collider, if this GameObject is performing a Move().

Character Controller Controls

`SimpleMove(Vector3)` and `Move(Vector3)`: Move the character.

Both functions move the GameObject and react to collisions (sliding, if possible). Unity recommends not to call `Move()` or `SimpleMove()` more than once per frame. These are their differences:

Function	Axis	Affected by gravity	Returns
`SimpleMove()`	*XZ*	YES	`isGrounded`
`Move()`	*XYZ*	NO	`CollisionFlags`

Physics Best Practices

There are a few things to be aware of:

Layers and collision matrix: By default, all objects are assigned to the *default* layer. Therefore, everything can collide, by default, with everything, which is quite inefficient. **Hint**: Assign objects to layers and disable those collisions in the matrix that are not meant to happen.

Physics Best Practices – Raycasting

Raycasting is useful, but costly.

- Use the least amount of rays possible.
- Set a distance limit, if possible.
- Don't use Raycasts inside a FixedUpdate() function.
- Very complex colliders (i.e. mesh colliders) are very expensive when raycasting against them. Use an approximation with a simpler collider instead.
- Specify a layer mask to limit the number of collisions the raycast checks.
- A more efficient way of specifying a layer mask is with *bit operators*: if you want a ray to hit an object which is on layer which id is 10, what you should specify is `layerMask = 1 << 10`. If you want for the ray to hit everything except what is on layer 10 then simply use the *bitwise complement operator* ($\sim$) which reverses each bit on the the bitmask.

Physics Best Practices – RigidBody

- GameObjects without a RigidBody component are considered *static colliders*. It is extremely inefficient to attempt to move *static colliders*, as it forces the physics engine to recalculate the physical world all over again.
- Making one RigidBody have greater Mass than another does not make it fall faster in free fall. Use Drag for that.
- If you are directly manipulating the Transform component of your object but still want collisions and trigger messages, attach a RigidBody and make it Kinematic.
- You cannot make an object stop rotating just by setting its Angular Drag to infinity.

Quiz!

https://forms.office.com/r/XPaqhV9kzZ