CE318: High-level Game Development

Lecture 3: 3D Games: Models and Physics

Diego Perez

dperez@essex.ac.uk
Office 3A.527

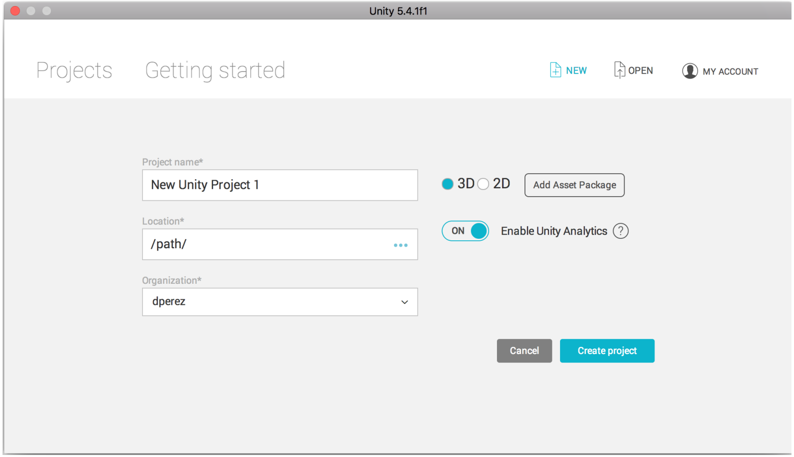
2016/17

Outline

Outline

The 3D Mode

In Unity 5.4, when you start a new project, you can select if you want to do a 2D or a 3D game. This sets the editor in the 2D / 3D mode:



Project name*: New Unity Project 1

Location*: /path/

Organization*: dperez

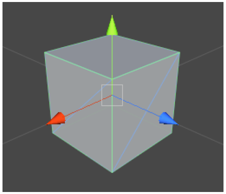
[✓ 3D] 2D

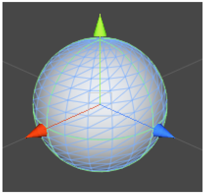
ON Enable Unity Analytics

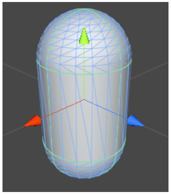
By default, the *Scene View* will be set to 3D view, although it is possible to switch to 2D view at any time.

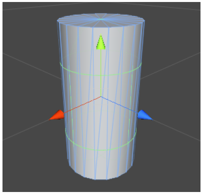
3D Game Objects (Primitive Meshes)

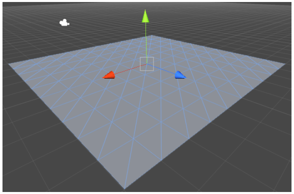
Unity has some pre-defined primitive meshes:

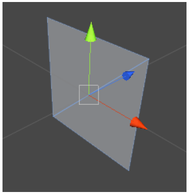

Cube


Sphere


Capsule

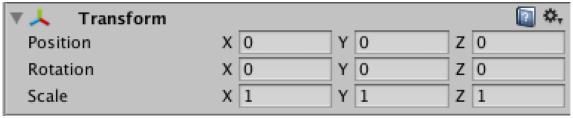

Cylinder


Plane


Quad

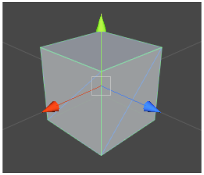
3D Transforms

All 3D game objects have a Transform component:

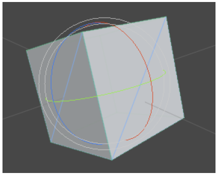


Transform						
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

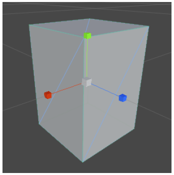
Translation, Rotation and Scale can be set in the transform or applied directly on the *Scene View* using the control gizmos.



Translation



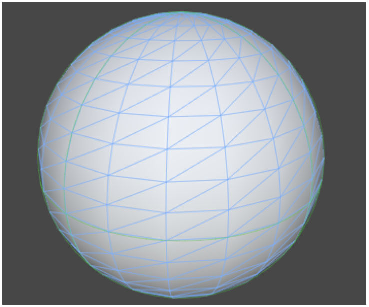
Rotation



Scale

Vertices, Triangles, Meshes

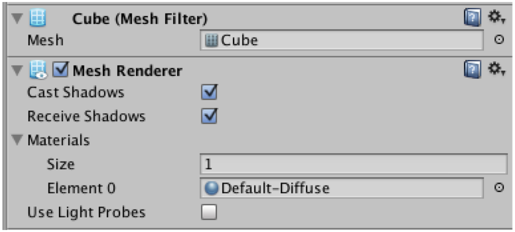
A **mesh** is a collection of 3D points (**vertices**) that, when combined together, define a physical presence of an object. Every mesh is made up of a series of **triangles**, given by its vertices.



Triangles are used because:

- 3 points determine a unique triangle with sides that don't cross each other.
- 3 points determine a triangle in a unique plane (flat shape).

Mesh Filter and Renderer



Cube (Mesh Filter)

Mesh	Cube

Mesh Renderer

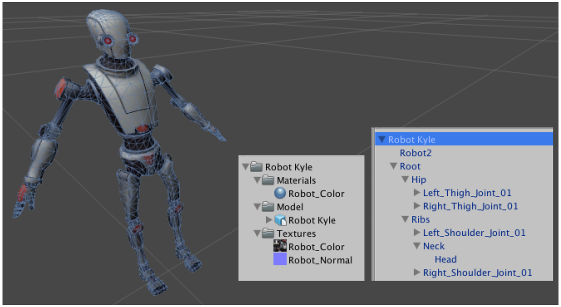
Cast Shadows	✓
Receive Shadows	✓
Materials	
Size	1
Element 0	Default-Diffuse
Use Light Probes	☐

Mesh Filter: The Mesh Filter takes a mesh from your assets and passes it to the Mesh Renderer for rendering on the screen.
Mesh Renderer: Takes the geometry of the object from the mesh filter and renders it at the position defined by the objects Transform component.

- Cast Shadows: If enabled, this Mesh will create shadows when a shadow-creating light shines on it.
- Receive Shadows: If enabled, this Mesh will display any shadows being cast upon it.
- Materials : A list of Materials to render model with (see later).
- Use Light Probes: Enable probe-based lighting for this mesh (see further lectures).

Models (1/2)

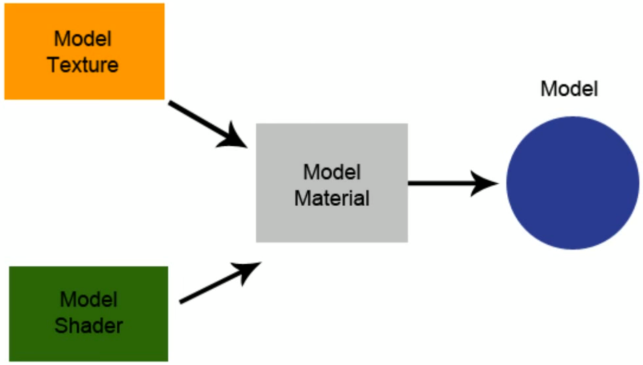
A **model** is a mesh with its applied material, texture and shader.



Supported formats: Importing meshes into Unity can be achieved from two main types of files:

- Exported 3D file formats, such as .FBX or .OBJ.
- Proprietary 3D application files, such as .Max and .Blend file formats from 3D Studio Max or Blender for example.
- Unity can read .FBX, .dae (Collada), .3DS, .dxf and .obj files.

Models (2/2)

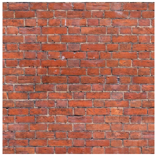


- A model has a Model Material applied to it.
- A **material** can be seen as the skin of the mesh.
- Every mesh needs a material in order to be seen on the screen.
- The model material (saved in a file) has two parts: the texture and the shader.

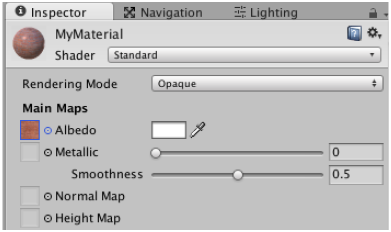
Materials *

Unity provides a default material for every mesh (simple, plain grey material).

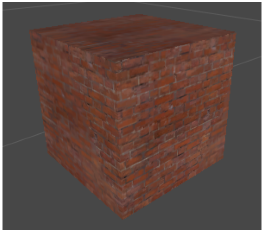
One of its basic parameters is a texture. A texture is a simple image file:



We can create a new material in Unity and assign the texture to it. Then, it is possible to apply the new material to the object:



MyMaterial
Shader Standard
Rendering Mode Opaque
Main Maps
Albedo
Metallic 0
Smoothness 0.5
Normal Map
Height Map



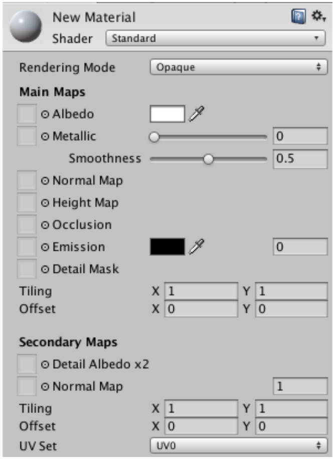
Shaders

Shaders in Unity are used through Materials. The shader code tells the material which properties it takes (colors, textures, transparency, etc.).

The default shader is the *Standard* Shader, used in the default material. It is a very versatile shader that can be configured in multiple ways.

There are three sections in the shader material:

1. **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 it's reflectivity.



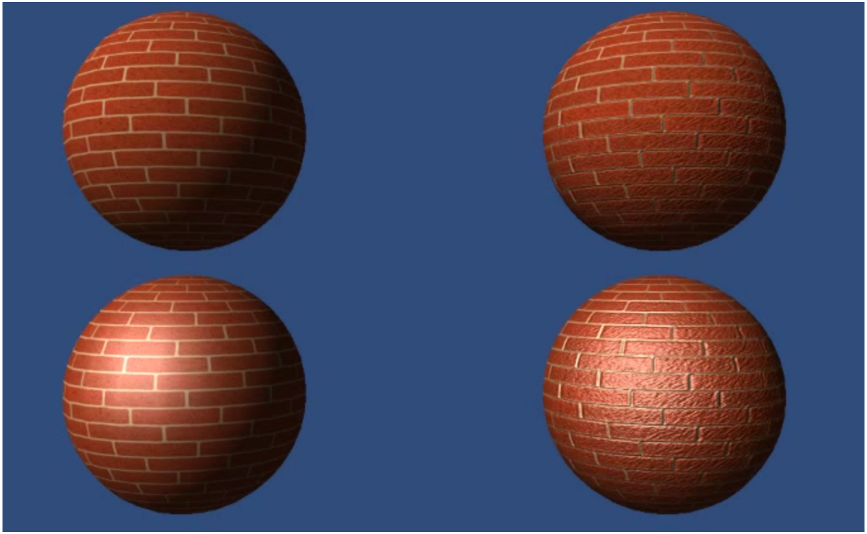
New Material		
Shader	Standard	
Rendering Mode	Opaque	
Main Maps		
Albedo		
Metallic		0
Smoothness		0.5
Normal Map		
Height Map		
Occlusion		
Emission		0
Detail Mask		
Tiling	X 1	Y 1
Offset	X 0	Y 0
Secondary Maps		
Detail Albedo x2		
Normal Map		1
Tiling	X 1	Y 1
Offset	X 0	Y 0
UV Set	UV0	

The Standard Shader

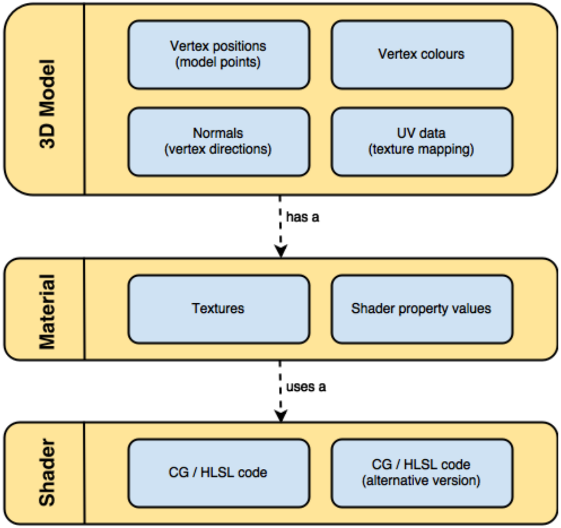
2. **Main Maps:** Properties defined by texture maps:
    - Albido: combination of an optional texture and a colour value to tint the texture (white: unaffected).
    - Metallic: “metalness” of the material’s surface, defined by a texture or a slider. The texture colour defines how metallic each point is (0 red for totally metallic, 255 red for no metallic).
        - Smoothness specifies how diffuse/sharp reflections are ($0 \rightarrow 1$).
        - Alpha channel of the texture defines the smoothness.
    - Normal map: defines apparent bumpiness of the surface.
    - 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.
3. Secondary maps: to 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.

Shaders



The Whole Picture


3D Model
Vertex positions (model points)
Vertex colours
Normals (vertex directions)
UV data (texture mapping)
has a
Material
Textures
Shader property values
uses a
Shader
CG / HLSL code
CG / HLSL code (alternative version)

Outline

3D Physics

Unity has a separated physics engine for all 3D interactions in the game. The components of the physics 3D engine in Unity are:

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

We'll also look at Physic 3D Materials and Raycasting in this lecture.

3D Rigidbody (1/3)

A **Rigidbody** is the main component that enables physical behaviour for an object. With a Rigidbody attached, the object will immediately respond to gravity.

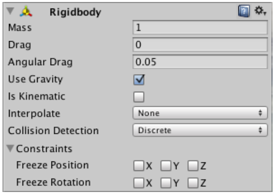
Since a Rigidbody component takes over the movement of the object it is attached to, you shouldn't try to move it from a script by changing the Transform properties such as position and rotation. Instead, you should apply forces to push the object and let the physics engine calculate the results.

The Rigidbody component has a property called Is Kinematic which will **remove it** from the control of the physics engine and allow it to be moved kinematically from a script.

Sleeping

Once a rigidbody is moving at less than a certain minimum linear or rotational speed, the physics engine will assume it has come to a halt. When this happens, the object will not move again until it receives a collision or force and so it will be set to **sleeping mode**. This optimisation means that no processor time will be spent updating the rigidbody until the next time it is set in motion again.

3D Rigidbody (2/3)



Rigidbody	
Mass	1
Drag	0
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	☐
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	☐X ☐Y ☐Z
Freeze Rotation	☐X ☐Y ☐Z

Parameters:

- Mass: The mass of the object (arbitrary units). You should not make masses more or less than 100 times that of other Rigidbodies.
- Drag: How much air resistance affects the object when moving from forces. 0 means no air resistance, and infinity makes the object stop moving immediately.
- Angular Drag: How much air resistance affects the object when rotating from torque. 0 means no air resistance.
- Use Gravity: If enabled, the object is affected by gravity.
- Is Kinematic: If enabled, the object will not be driven by the physics engine, and can only be manipulated by its Transform.

3D Rigidbody (3/3)

- Interpolate:
  - None: no interpolation (default).
  - Interpolate: Transform smoothed based on the position of the previous frames.
  - Extrapolate: Transform smoothed based on the current velocity.
- Collision Detection:
  - *Discrete:* Normal collision detection.
  - *Continuous:* prevents fast-moving colliders from passing each other. This may happen when using normal (Discrete) collision detection, when an object is one side of a collider in one frame, and already passed the collider in the next frame (supported for Box-, Sphere- and CapsuleColliders).
- Continuous Dynamic: Use continuous collision detection against objects set to Continuous and Continuous Dynamic Collision.
- Constraints: Restrictions on the Rigidbodys motion:
  - Freeze Position: Stops the Rigidbody moving in the world X, Y and Z axes selectively.
  - Freeze Rotation: Stops the Rigidbody rotating around the world X, Y and Z axes selectively.

3D Rigidbody in code

Scripts within a game object with a rigidbody attached will be able to access this rigidbody component through GetComponent<Rigidbody>() call.

It is possible to move an object with a rigidbody by changing its position, linear, angular velocity, etc. For instance:

```
//Get the rigidbody
Rigidbody rigidbody = GetComponent<Rigidbody>();

//This sets the angular velocity of the rigidbody.
rigidbody.velocity = transform.forward * speed;

//This sets the angular velocity of the rigidbody.
rigidbody.angularVelocity = new Vector3(1f,0.5f,2f);

//This sets the position of the rigidbody.
rigidbody.position += new Vector3(0f,0.5f,0f);
```

You can also access the other variables of the rigidbody component, such as `drag`, `angular drag`, `gravity` or if it is a `kinematic` object.

3D Colliders (1/3)

Collider components define the shape of an object for the purposes of physical collisions. A collider, which is invisible, does not need to be the exact same shape as the object's mesh and in fact, a rough approximation is often more efficient and indistinguishable in gameplay.

Any number of these can be added to a single object to create compound colliders. With careful positioning and sizing, compound colliders can often approximate the shape of an object quite well while keeping a low processor overhead.

Further flexibility can be gained by having additional colliders on child objects. However, you should be sure that there is only one Rigidbody and this should be placed on the root object in the hierarchy.

A good general rule is to use mesh colliders for scene geometry and approximate the shape of moving objects using compound primitive colliders.

3D Colliders (2/3)

Colliders interact with each other differently depending on how their Rigidbody components are configured. The three important *configurations* are:

Static Collider: GameObject with a Collider but no Rigidbody.

Used for level geometry which always stays at the same place and never moves around. Incoming rigidbody objects will collide with the static collider but will not move it.

The physics engine assumes that static colliders never move or change and can make useful optimizations based on this assumption.

Consequently, static colliders should not be disabled/enabled, moved or scaled during gameplay.

3D Colliders (3/3) *

Rigidbody Collider: GameObject with a Collider and a normal, non-kinematic Rigidbody attached.

Rigidbody colliders are fully simulated by the physics engine and can react to collisions and forces applied from a script. They can collide with other objects (including static colliders) and are the most commonly used Collider configuration in games that use physics.

Kinematic Rigidbody Collider: GameObject with a Collider and a kinematic Rigidbody attached.

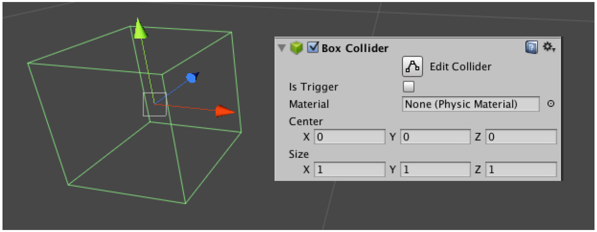
You can move a kinematic rigidbody object from a script by modifying its Transform Component but it will not respond to forces like a non-kinematic rigidbody.

Kinematic rigidbodies should be used for colliders that can be moved or disabled/enabled occasionally but that should otherwise behave like static colliders.

An example of this is a sliding door that should normally act as an immovable physical obstacle but can be opened when necessary.

3D Colliders - Box Collider

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.

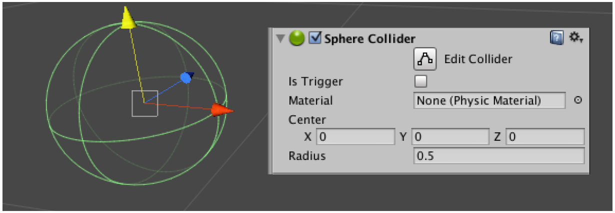


Is Trigger ☐
Material: None (Physic Material)
Center: X 0, Y 0, Z 0
Size: X 1, Y 1, Z 1

- Is Trigger: If enabled, this Collider is used for triggering events, and is ignored by the physics engine.
- Material: Reference to the Physics Material that determines how this Collider interacts with others.
- Center: The position of the Collider in the objects local space.
- Size: The size of the Collider in the X, Y, Z directions.

3D Colliders - Sphere Collider

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.

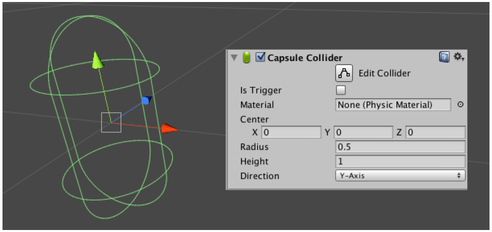


Is Trigger	☐
Material	None (Physic Material)
Center	X 0 Y 0 Z 0
Radius	0.5

- In Trigger: If enabled, this Collider is used for triggering events, and is ignored by the physics engine.
- Material: Reference to the Physics Material that determines how this Collider interacts with others.
- Center: The position of the Collider in the objects local space.
- Radius: The radius of the sphere.

3D Colliders - Capsule Collider

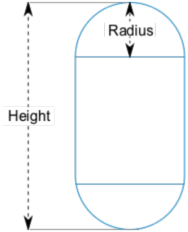
Capsule Collider: The Capsule Collider is made of two half-spheres joined together by a cylinder. It is the same shape as the Capsule primitive.



Capsule Collider

Edit Collider

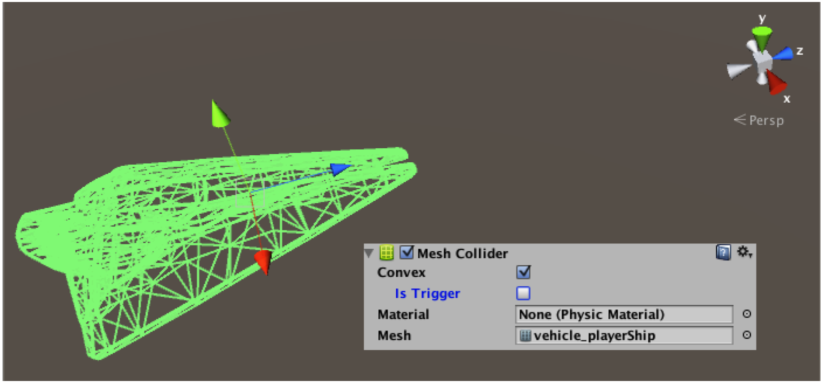
Is Trigger	☐
Material	None (Physic Material)
Center	X 0 Y 0 Z 0
Radius	0.5
Height	1
Direction	Y-Axis



- Is Trigger: If enabled, this Collider is used for triggering events, and is ignored by the physics engine.
- 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.

3D Colliders - Mesh Collider (1/2)

Mesh Collider: The Mesh Collider takes a Mesh Asset and builds its Collider based on that mesh. It is far more accurate for collision detection than using primitives for complicated meshes. By default, Mesh Colliders do not collide with other Mesh Colliders. Mesh Colliders are more computationally expensive than primitive collider types, so it is best to use them sparingly.



Mesh Collider

Convex	✓
Is Trigger	☐
Material	None (Physic Material)
Mesh	vehicle_playerShip

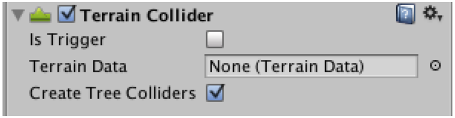
3D Colliders - Mesh Collider (2/2)

Properties of the Mesh Collider:

- 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.
  - Is Trigger: If enabled, this Collider is used for triggering events, and is ignored by the physics engine. Trigger functionality in mesh colliders is only possible if the collider is marked as convex.
- Material: Reference to the Physics Material that determines how this Collider interacts with others.
- Mesh: Reference to the Mesh to use for collisions.

3D Colliders - Terrain Collider

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

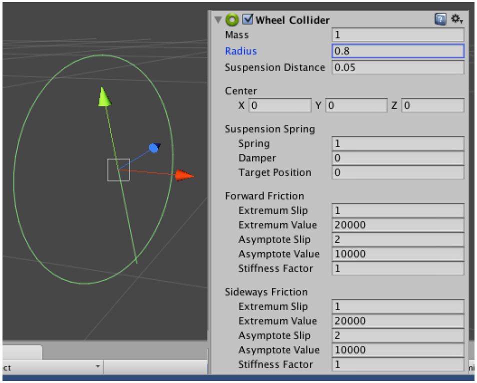

Terrain Collider

Is Trigger	☐
Terrain Data	None (Terrain Data)
Create Tree Colliders	✓

- Is Trigger: If enabled, this Collider is used for triggering events, and is ignored by the physics engine.
- Terrain Data: The terrain data.
- Create Tree Colliders: When selected Tree Colliders will be created.

3D Colliders - Wheel Collider

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.



Wheel Collider

Mass	1
Radius	0.8
Suspension Distance	0.05
Center	X 0 Y 0 Z 0
Suspension Spring	
Spring	1
Damper	0
Target Position	0
Forward Friction	
Extremum Slip	1
Extremum Value	20000
Asymptote Slip	2
Asymptote Value	10000
Stiffness Factor	1
Sideways Friction	
Extremum Slip	1
Extremum Value	20000
Asymptote Slip	2
Asymptote Value	10000
Stiffness Factor	1

For all parameter descriptions, check:

`http://docs.unity3d.com/Manual/class-WheelCollider.html`

Colliders as Triggers

Colliders set as **triggers** will **not** participate in physical collisions. If a collision happens with a trigger 3D collider, the following messages will be sent:

- `void OnTriggerEnter (Collider other)`: Sent when another collider enters a trigger collider attached to the game object.
- `void OnTriggerExit (Collider other)`: Sent when another collider leaves a trigger collider attached to the game object.
- `void OnTriggerStay (Collider other)`: Sent once per frame for every Collider (other) that is touching the trigger.

However, a 3D collider not set up as a trigger will participate in physical collisions, sending the following messages:

- `void OnCollisionEnter (Collision other)`: Sent when an incoming collider makes contact with this object's collider.
- `void OnCollisionExit (Collision other)`: Sent when a collider on another object stops touching the object's collider.
- `void OnCollisionStay (Collision other)`: Sent each frame where a collider on another object is **touching** the object's collider.

Forces

A Force is any interaction which tends to change the motion of an object. In Unity, forces are represented as vectors, and can be linear (known simply as Force) or rotational (known as Torque). Also, they can be applied to entire objects or upon a specific point.

The biggest difference between manipulating the Transform versus the Rigidbody is the use of **forces**: rigidbodies can receive forces and torque, but Transforms cannot. Adding forces/torque to the Rigidbody will actually change the objects position and rotation of the Transform component.

Important: Changing the Transform while using physics could cause problems with collisions and other calculations.

In a script, the *FixedUpdate* function is recommended as the place to apply forces and change Rigidbody settings. The reason for this is that physics updates are carried out in measured time steps that don't coincide with the frame update. *FixedUpdate* is called immediately before each physics update and so any changes made there will be processed directly.

Forces are dampened by the rigidbody's *drag* property.

AddForce *

```
public void AddForce(Vector3 force, ForceMode mode = ForceMode.Force);
public void AddForce(float x, float y, float z, ForceMode mode = ForceMode.Force);
```

`Vector3 force` (or `float` variables `x,y,z`) determine the direction and magnitude of the force to be applied. Examples:

```
rigidbody.AddForce(Vector3.up * 10);
rigidbody.AddForce(0, 10, 0);
```

`ForceMode` specifies how the force is applied:

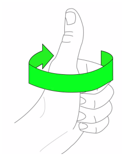
- ForceMode.Force: Continuous changes that are affected by mass (movement depends on the mass of the object).
- ForceMode.Acceleration: Add a continuous acceleration to the rigidbody, ignoring its mass (movement **does not** depend on the mass of the object).
- ForceMode.Impulse: Add an instant force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

AddTorque *

Torque is the tendency of a force to rotate an object about an axis. *AddTorque* adds a torque force to a Rigidbody. As a result the rigidbody will start spinning around the *torque* axis.

```
public void AddTorque(Vector3 torque, ForceMode mode = ForceMode.Force);

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

The force is applied following the left hand screw rule:



```
public float amount = 50f;
void FixedUpdate ()
{
    float h = Input.GetAxis("Horizontal") * amount ;
    float v = Input.GetAxis("Vertical") * amount;

    //rigidbody.AddForce(transform.up * v);
    //rigidbody.AddForce(transform.right * h);

    rigidbody.AddTorque(transform.up * h);
    rigidbody.AddTorque(transform.right * v);
}
```

Other Forces to Add

AddRelativeForce: Adds a force to the rigidbody relative to its **local** coordinate system.

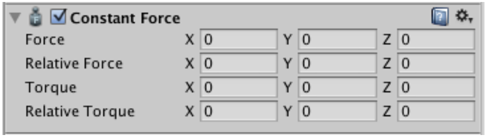
```
public void AddRelativeForce(Vector3 force, ForceMode mode = ForceMode.Force);
public void AddRelativeForce(float x, float y, float z, ForceMode mode=ForceMode.Force);
```

AddForceAtPosition: Applies *force* at *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. This is most commonly used for explosions.

```
public void AddForceAtPosition(Vector3 force, Vector3 position, ForceMode mode = ForceMode
.Force);
```

Constant Force

ConstantForce: Constant Force is a component that can be added to any game object 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.

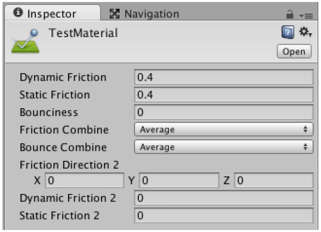

Constant Force

Force	X 0	Y 0	Z 0
Relative Force	X 0	Y 0	Z 0
Torque	X 0	Y 0	Z 0
Relative Torque	X 0	Y 0	Z 0

- 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.
- Relative Torque: The vector of a torque, applied in local space. The object will begin spinning around this vector. The longer the vector is, the faster the rotation.

Physics Materials (1/2)

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.



TestMaterial

Dynamic Friction	0.4
Static Friction	0.4
Bounciness	0
Friction Combine	Average
Bounce Combine	Average
Friction Direction 2	X 0 Y 0 Z 0
Dynamic Friction 2	0
Static Friction 2	0

- 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.
- Bounciness: How bouncy is the surface? A value of 0 will not bounce. A value of 1 will bounce without any loss of energy.

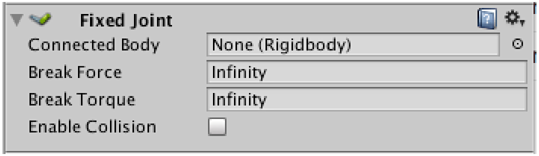
Physics Materials (2/2)

- 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 (1/2)

Joints are used to restrict one object's movement so it depends on another object. This dependence is implemented by physics, instead of parenting in an object hierarchy. There are different types of joints:

Fixed Joint: used to have two objects stuck together permanently or temporarily. Both objects must have a Rigidbody component.



Fixed Joint

Connected Body	None (Rigidbody)
Break Force	Infinity
Break Torque	Infinity
Enable Collision	☐

- Connected Body: Optional reference to the Rigidbody that the joint is dependent upon. If not set, the joint connects to the world.
- Break Force : The force that needs to be applied for this joint to break.
- Break Torque: The torque that needs to be applied for this joint to break.
- Enable Collision: When checked, this enables collisions between bodies connected with a joint.

Joints (2/2)

Spring Joint: groups together two Rigidbodies, constraining them to move like they are connected by a spring.

Hinge Joint: groups together two Rigidbodies, constraining them to move like they are connected by a hinge. It is perfect for doors, but can also be used to model chains, pendulums, etc.

Character Joint: groups together two Rigidbodies, limiting their rotation angles to each other. They are mainly used for Ragdoll effects. Ragdoll is used as a type of procedural animation used to replace static death animations.

Configurable Joint: fully customizable joint, movement/rotation restriction and movement/rotation acceleration of the attached Rigidbodies.

Raycasting (1/3)

A **ray** is an infinite line starting at origin and going in some direction. Unity has a class `Ray` with two attributes: `Vector3 origin`, the origin of the ray; and `Vector3 direction`, the direction of the ray.

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, RaycastHit hitInfo, float distance = Mathf.Infinity,
 int layerMask = DefaultRaycastLayers);
```

- ray: The starting point and direction of the ray.
- distance: The length of the ray.
- hitInfo: If true is returned, hitInfo will contain more information about where the collider was hit.
- layerMask: A Layer mask that is used to selectively ignore colliders when casting a ray.

Physics.Raycast returns a `bool` which value is `true` if the ray intersects a collider, `false` otherwise. If a collider is hit (i.e. the function returns `true`), information about the collision is returned in *hitInfo*.

Raycasting (2/3)

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.

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"UserLayer" and "UserLayer".

Raycasting (3/3)

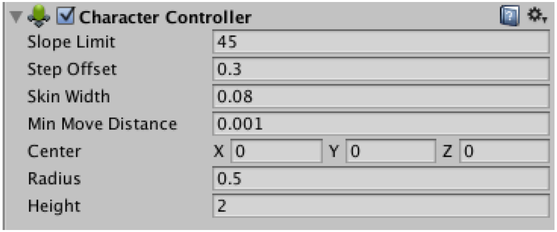
```
RaycastHit shootHit;
float range = 100f;
LayerMask shootable = LayerMask.GetMask ("Shootable");

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

LineRenderer gLine = GetComponent <LineRenderer> ();
gunLine.SetPosition (0, transform.position);

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

   // Set the second position of the line renderer to the point hit.
   gLine.SetPosition(1, shootHit.point);
}else{
   // If there was no hit, set the second position of the line
       renderer to the fullest extent of the gun's range.
   gLine.SetPosition(1, shootRay.origin + shootRay.direction * range);
}
```

A Special Component: Character Controller (1/2)

Character Controller is a *special* component available for game objects. 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 Capusle Collider, hence it reacts to collisions.


Character Controller

Slope Limit	45
Step Offset	0.3
Skin Width	0.08
Min Move Distance	0.001
Center	X 0 Y 0 Z 0
Radius	0.5
Height	2

A Special Component: Character Controller (2/2)

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.
- `SimpleMove(Vector3)` and `Move(Vector3)`: Move the character (see below).
- `CharacterController.OnControllerColliderHit`: called when colliding with another collider, if this game object is performing a Move().

Both functions move the game object 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	Speed in
`SimpleMove()`	*XZ*	YES	`isGrounded`	*meters per second*
`Move()`	*XYZ*	NO	`CollisionFlags`	*meters*

Physics Best Practices (1/2)

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.

Raycast: Raycasting is very powerful and useful, but it is an expensive operation for the physics engine. **Hints**:

- Use the least amount of rays possible.
- Set a distance limit, if possible.
- Dont use 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 (2/2)

Rigidbody hints:

- Game objects 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.

Outline

UROP & Lab Preview

UROP: Undergraduate Research Opportunities Programme (UROP)

- Useful information:

  `https://www.essex.ac.uk/urop/students/useful-forms.aspx`
- Web Game Engine placement:
  - **General Video Game Artificial Intelligence: Playing and Data Collection on the Web**:
  - `https://www.essex.ac.uk/urop/students/placement-details.aspx?ID=198`

During this and next week's lab you will create a 3D game:

- The game will consist of a 3D space shooter, although it only expands in two dimensions, as it is seen from above.
- You will learn how to set a project up, create a scene, place game objects, process input to move the player, etc.

Next lecture: Cameras, Audio, Lights and Shadows.