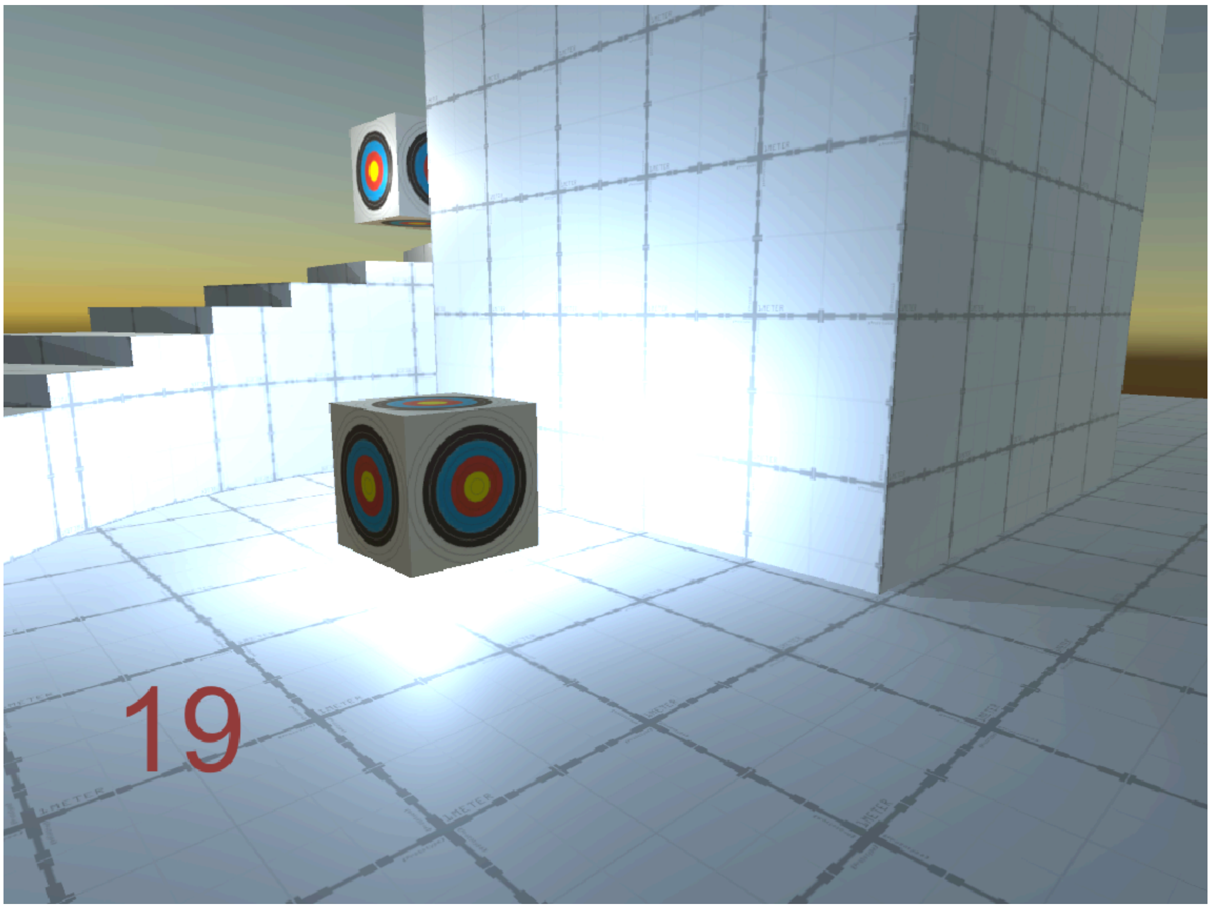
COMP4002/G54GAM Lab Exercise 03 – First Person Shooting Gallery

19/02/19

Note – this exercise does require you to create a Unity account so that an add-on can be downloaded from the asset store.

This lab exercise involves creating a first-person shooting-gallery game – however it reuses some of the concepts that have been seen already, while also introducing some new ones. The instructions are a little more sparse than before.



The game is going to have the following features:

- A character that can strafe left and right, and move forwards and backwards using the keyboard
- A first-person view from the character's perspective that can be moved using the mouse ("mouselook")
- A series of targets that the player must shoot by firing a projectile
- A timer that counts down until all the targets have been hit
- A head-up-display (HUD) that shows the time remaining

The majority of these features variations on what has been seen in the previous exercises – objects interacting with one another via function calls, a player controller and a game controller.

The workflow here is similar to before:

- Create a basic environment
- Create a character that the player can move around
- Create something for the player to do (shoot at targets)

But also…

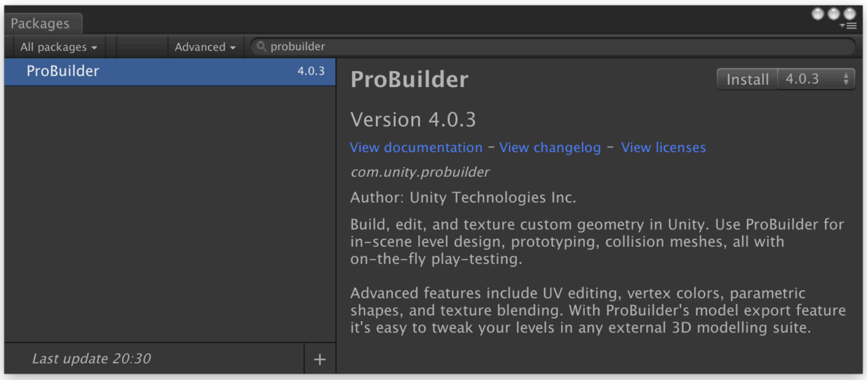
- Start decrementing a timer at the start of the game
- Do something when a target is hit
  - Add some additional time to the timer
- Provide an **end goal**
  - Win the game when all targets are hit
  - Lose the game when the time runs out

Creating the Environment

Instead of using sprites or an image as a background, this game will make use of a 3D environment with a floor, walls and other features such as stairs and ramps as you see fit. The workflow for constructing a 3D environment, or terrain geometry, varies between game engines. Often the rough geometry of the environment is blocked out (or "greyboxed") using basic shapes before being decorated with textures and 3D mesh details. Alternatively, the terrain can be constructed entirely in a modelling package and then imported as a collection of objects. Unreal engine has support for BSP brush modelling. Unity has some support for this kind of modelling using a package called *ProBuilder*, which provides a basic equivalent to the conventional style of BSP modelling.

Integrating ProBuilder

On your own machine / latest version of Unity. ProBuilder has recently been acquired and integrated as a *package* into Unity. To use it, go to Window->Package Manager to open the package manager dialogue. Search for "ProBuilder", then click Install, and it will be added to your project under the Tools->ProBuilder menu.



ProBuilder

Version 4.0.3

View documentation - View changelog - View licenses

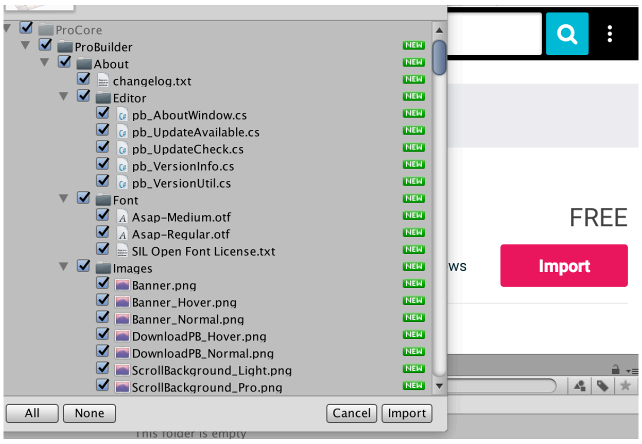
com.unity.probuilder

Author: Unity Technologies Inc.

Build, edit, and texture custom geometry in Unity. Use ProBuilder for in-scene level design, prototyping, collision meshes, all with on-the-fly play-testing.

Advanced features include UV editing, vertex colors, parametric shapes, and texture blending. With ProBuilder's model export feature it's easy to tweak your levels in any external 3D modelling suite.

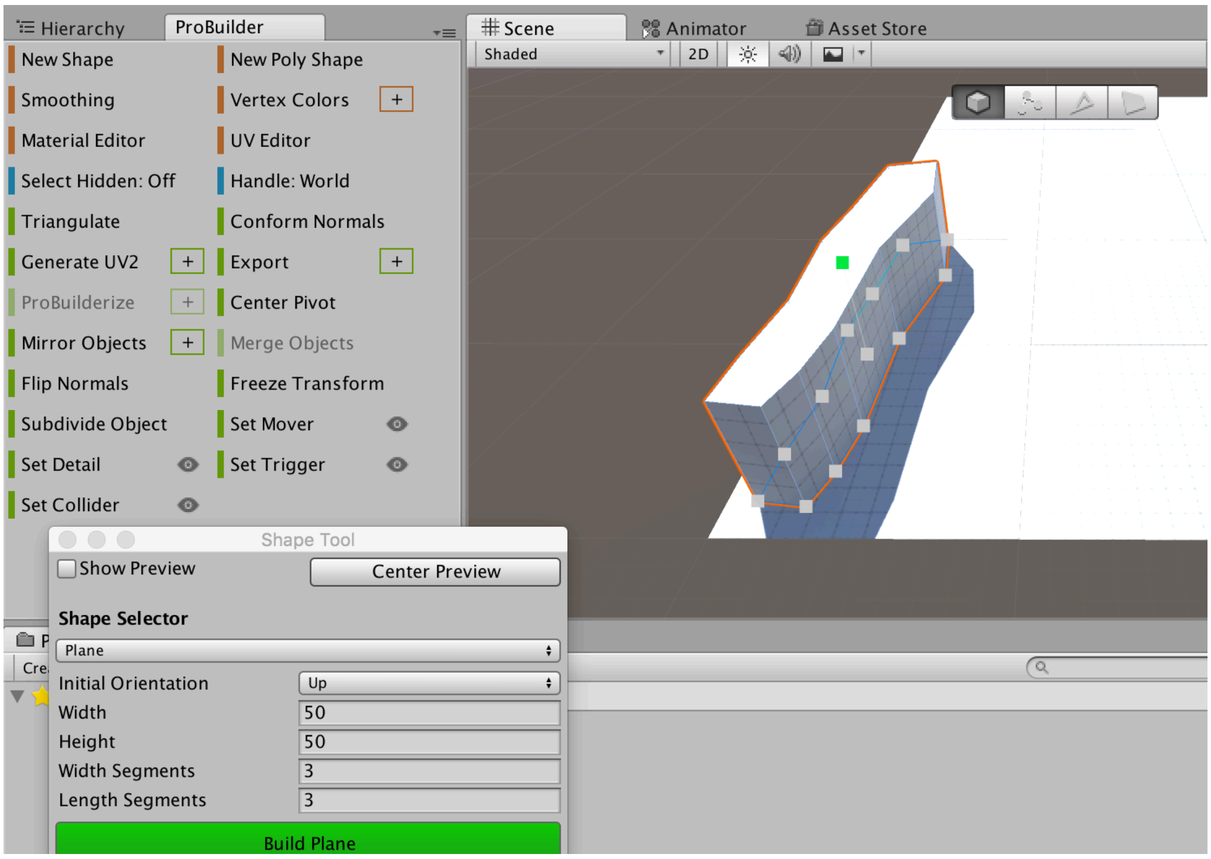
On the machines in A32. Open the Unity Asset Store via Window->Asset Store. Search for and open the Probuilder 2.x, then click Download. This will prompt you to log into Unity. Now click Import, followed by Import in the Unity dialogue that appears. This will integrate ProBuilder into your project, where again it will be available under Tools->ProBuilder.



```
ProCore
  ProBuilder                      NEW
    About                         NEW
      changelog.txt               NEW
      Editor                      NEW
        pb_AboutWindow.cs         NEW
        pb_UpdateAvailable.cs     NEW
        pb_UpdateCheck.cs         NEW
        pb_VersionInfo.cs         NEW
        pb_VersionUtil.cs         NEW
      Font                        NEW
        Asap-Medium.otf           NEW
        Asap-Regular.otf          NEW
        SIL Open Font License.txt NEW
      Images                      NEW
        Banner.png                NEW
        Banner_Hover.png          NEW
        Banner_Normal.png         NEW
        DownloadPB_Hover.png      NEW
        DownloadPB_Normal.png     NEW
        ScrollBackground_Light.png NEW
        ScrollBackground_Pro.png  NEW
```

Note that there are a great many assets available (some for free, some paid) in the asset store – including models, textures, sounds, scripts etc.

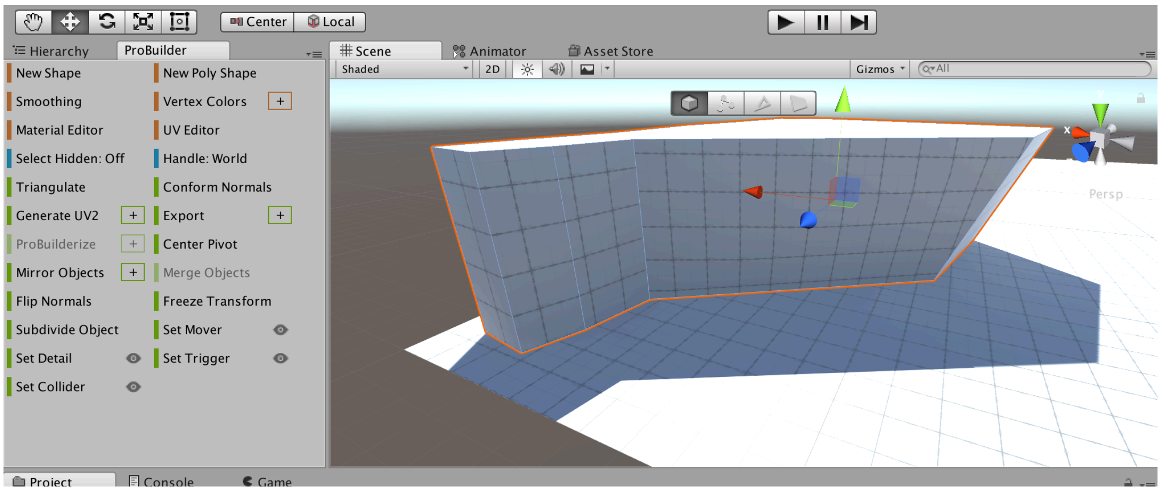
Open the ProBuilder window (Tools->ProBuilder->ProBuilder Window). You might find it useful to dock this tab somewhere convenient rather than leaving it floating.



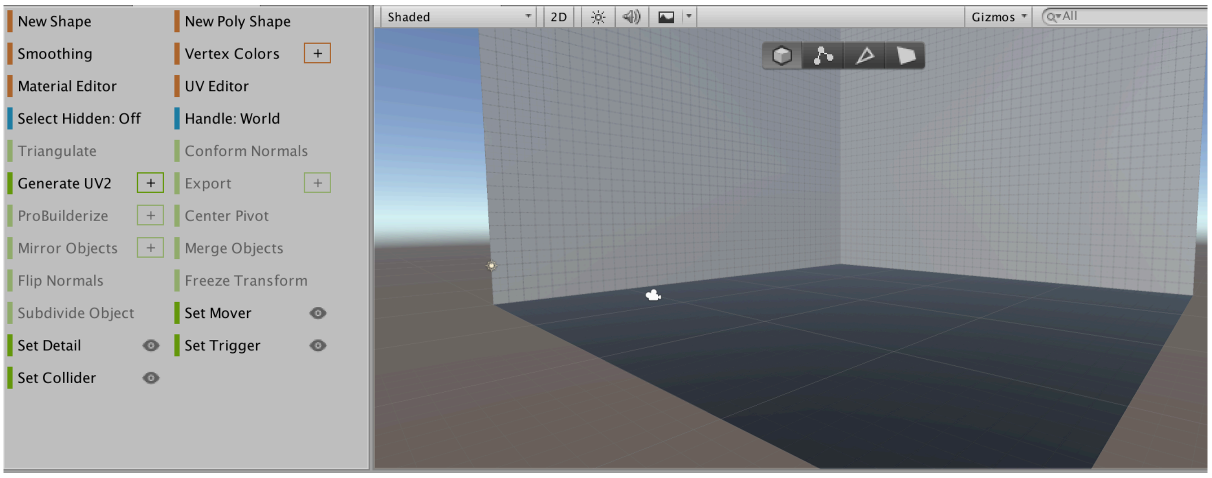
Shape Selector

	Plane
Initial Orientation	Up
Width	50
Height	50
Width Segments	3
Length Segments	3

Clicking “New Shape”, or Ctrl + Shift + K, brings up the Shape Tool, where you can select from a number of primitive geometry shapes to include – it is sufficient to just create a fairly large flat *Plane* to start with, but also experiment with the other shapes available. In particular, the Poly Shape allows you to define a polygon using a number of points, and then extrude this upwards into a solid shape. The icons that have appeared in the scene view allow you to select an entire shape, a vertex (a point on the object), an edge or a face, and manipulate this to modify the object.



You can also create an “indoor” space by creating a cube, or a poly shape, and clicking *Flip Normals*. This has the effect of turning the shape inside out – so the top becomes a ceiling, and a cube becomes an empty box.



ProBuilder is quite a powerful tool. A walkthrough of how to create a simple interior level is available here:

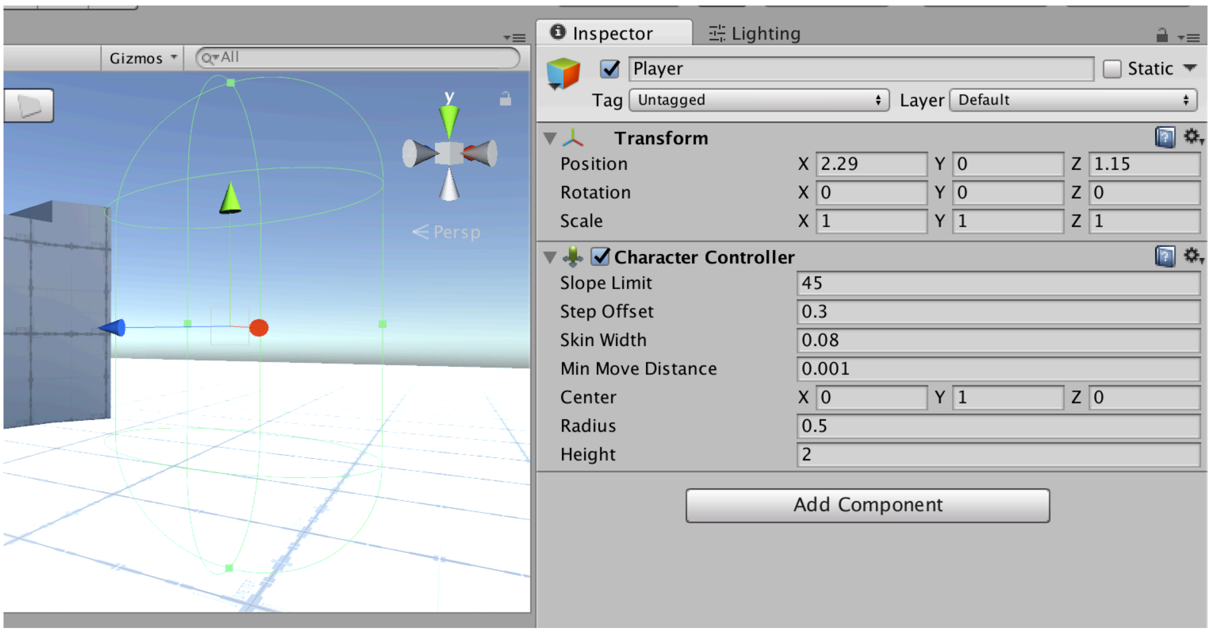
https://www.youtube.com/watch?v=dYBOBgfcTgY

FPS Character Movement

Create a new empty game object to act as the *Player* embodiment in the environment. As this is a first-person game (in that the player has a first-person perspective into the environment) the player’s character does not need to have a sprite, or a visible model.

Add a *CharacterController* component to the player object. This is a component that provides a moveable body, and allows simplified movement constrained by collisions, without us having to use a rigid body or apply forces:

https://docs.unity3d.com/ScriptReference/CharacterController.html



Player — Static
Tag: Untagged — Layer: Default

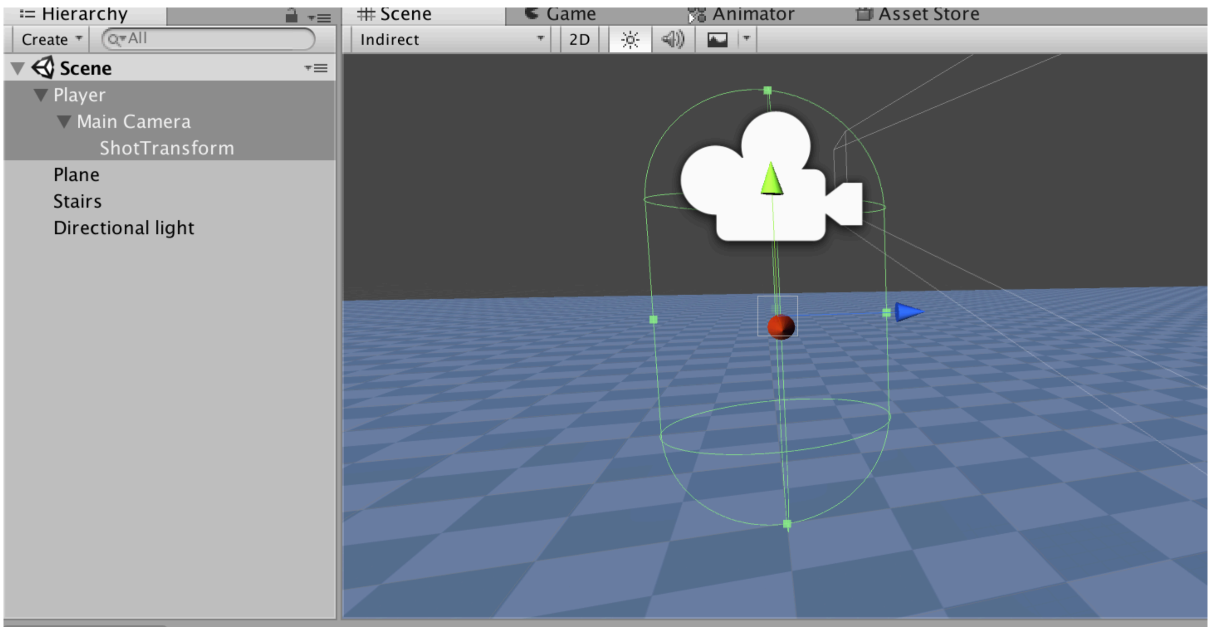
Transform

Position	X	2.29	Y	0	Z	1.15
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

Character Controller

Slope Limit	45
Step Offset	0.3
Skin Width	0.08
Min Move Distance	0.001
Center	X 0 Y 1 Z 0
Radius	0.5
Height	2

In the screenshot above the *Player* object is at 0 on the Y axis, and then the centre of the character controller has been raised to Y=1 so that the character controller body rests on the floor. This is not strictly necessary, but makes it easier to put the player exactly where we want on the level without having to worry about the height of the player.



Scene
- Player
  - Main Camera
    - ShotTransform
- Plane
- Stairs
- Directional light

To provide first-person perspective, the camera needs to move with the player. Drag the Main Camera in the scene and make it a *child* of the Player object. As with lab 01 also create an empty object and make *this* a child of the Camera. This will provide the transform with which to create bullet objects, and so the player will be able to shoot where they are currently looking.

The camera should be offset relative to the player – think about where the "eyes" of the player should be and accordingly change the height of the camera. Similarly, the shot transform object should be offset relative to the camera, and moved out in front of it.

As before, create a PlayerController script, which will take input to manipulate this setup:

```
float verticalVelocity = 0;

// Update is called once per frame
void Update()
{
    // rotate the player object about the Y axis
    float rotation = Input.GetAxis("Mouse X");
    transform.Rotate(0, rotation, 0);

    // rotate the camera (the player's "head") about its X axis
    float updown = Input.GetAxis("Mouse Y");
    Camera.main.transform.Rotate(updown, 0, 0);

    // moving forwards and backwards
    float forwardSpeed = Input.GetAxis("Vertical");

    // moving left to right
    float lateralSpeed = Input.GetAxis("Horizontal");

    // apply gravity
    verticalVelocity += Physics.gravity.y * Time.deltaTime;

    CharacterController characterController
        = GetComponent<CharacterController>();

    if (Input.GetButton("Jump") && characterController.isGrounded)
    {
        verticalVelocity = 5;
    }

    Vector3 speed = new Vector3(lateralSpeed, verticalVelocity, forwardSpeed);

    // transform this absolute speed relative to the player's current rotation
    // i.e. we don't want them to move "north", but forwards depending on where
    // they are facing
    speed = transform.rotation * speed;

    // what is deltaTime?
    // move at a different speed to make up for variable framerates
    characterController.Move(speed * Time.deltaTime);
}
```

In this exercise there is only one camera in the scene. Rather than retrieving the camera by inspecting the children of the object, a shortcut here is just to refer to "the main camera" via the static Camera class.

As before the vertical and horizontal inputs default to the WASD and cursor keys, and "Jump" is mapped to the space bar, but these can be remapped in the project settings (Edit->Project Settings->Input). The value for gravity is also a global project setting (Edit->Project Settings->Physics).

As it is, the player will move very slowly, and large mouse movements will be required to face in another direction. Vertical mouse movement is also inverted. Add appropriate public variables to the script and multiply the various speeds and rotations by these to achieve a responsive movement that you think is appropriate.

The shooting mechanic has a broadly similar implementation to lab 01, except will be a spherical object rather than a textured quad. Create a sphere (GameObject->3D Object->Sphere), and attach a *Rigidbody* component to it. As in lab 01 add a simple movement script, and furthermore destroy the object after an appropriate amount of time.

After making the new bullet object a prefab, it can be instantiated in a similar fashion again as lab 01, however as this is a 3D environment rather than top-down we now also need to consider the extra dimension afforded by the player looking up and down (pitch). The rotation of the camera is used to set the direction of the new bullet – this obviously also inherits the rotation (the yaw) of the character controller of which the camera is a child.

```
        if (Input.GetButton("Fire1") && Time.time > nextFire)
        {
            nextFire = Time.time + fireRate;

            Instantiate(shot,
                shotTransform.position,
                Camera.main.transform.rotation);
        }
```

Shooting Targets

The next step is to give the player something to shoot, but also to add an appropriate objective to give some meaning to the game. You may find it useful to refer back to previous exercises as the principles are broadly similar.

Destroy Bullets. Bullets should destroy themselves when they *collide* with any other object. As there is terrain in the scene that should not be destroyed, we'll have the bullet destroy itself, and have the object that it hit itself decide how to handle the collision.

```
    // script for the bullet
    private void OnCollisionEnter(Collision collision)
    {
        DestroyObject(gameObject);
    }
```

Destroyable Targets. A script (perhaps called *Destroyable*) should be attached to an appropriate object that you wish to use for the targets that the player must shoot. In the screenshot a simple cube has been with a target texture / material. The logic is slightly different from before. When the bullet collides with the target, the target destroys itself. The *OnDestroy* method is overridden, and when called itself calls *TargetDestroyed* on an appropriate Game Controller object, as in lab 01. This ordering is important, as the Game

Controller will count how many target objects are left, so this call must happen after the object is in fact destroyed, not just collided with.

```
// script for the target
public int timeBonus = 10;

private void OnCollisionEnter(Collision collision)
{
    // destroy this object
    DestroyObject(gameObject);
}

private void OnDestroy()
{
    // tell the game controller
    if (gameController != null)
    {
        gameController.TargetDestroyed();
    }
}
```

Keep Track of Time and Progress. Your game controller should keep track of the time, decreasing it by the elapsed time each update. When the time runs out, the player has lost. When each target is destroyed, check how many targets exist (by searching for objects that have your Destroyable script attached), and when none remain the player has won.

Extend this to give the player the appropriate time bonus for each target destroyed.

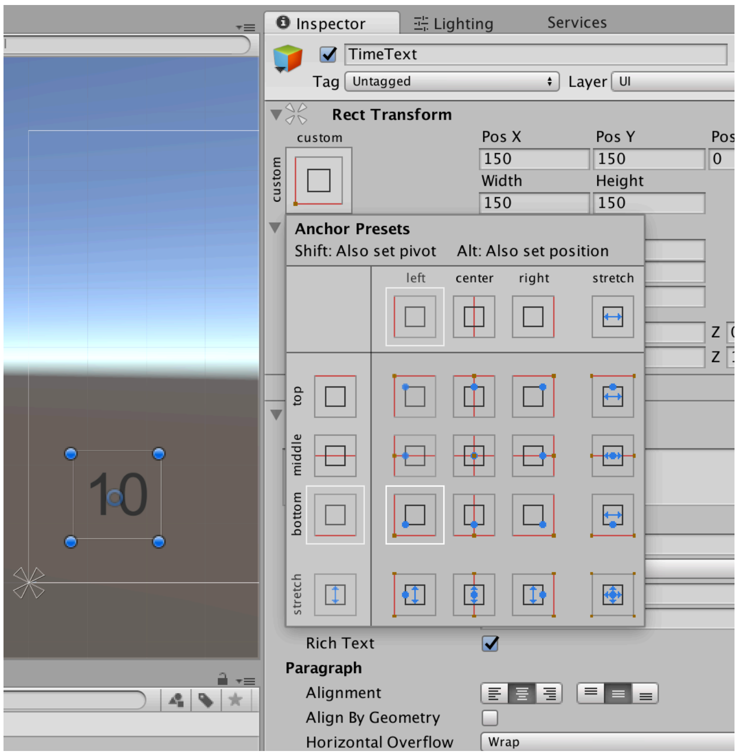
```
float timeLeft = 30.0f;

void Update()
{
    timeLeft -= Time.deltaTime;

    if (timeLeft < 0)
    {
        Debug.Log("game lost");
    }
}

public void TargetDestroyed()
{
    if(GameObject.FindObjectsOfType<Destroyable>().Length == 0)
    {
        Debug.Log("game won");
    }
}
```

Head Up Display

The final step is to present the time to the player so they can see how well they are doing, and this involves creating a head-up display, or HUD.

In the hierarchy create a *Text* object (GameObject->UI->Text). This will automatically create a *Canvas* to contain the Text object, which is a 2D user-interface widget. If you now run the game you can see the “New Text” floating over the player’s camera view.

Changing briefly to 2D mode in the scene view, select the canvas in the hierarchy and press F to zoom out the view so that it can be seen in its entirety. The extent of the canvas is automatically determined by the size and aspect ratio of the game window and scales to handle different screen resolutions. Multiple widgets, buttons, text fields and images can be placed on the canvas.



TimeText
Tag Untagged
Layer UI
Rect Transform
custom
Pos X 150
Pos Y 150
Pos Z 0
Width 150
Height 150
Rich Text ✓
Paragraph
Alignment
Align By Geometry
Horizontal Overflow Wrap

To handle different resolutions, rather than setting absolute positions for the widgets they are *anchored* to different parts of the screen and then offset accordingly. The screenshot shows the Text object, renamed TimeText, anchored 150 pixels away from the bottom left hand corner of the screen.

```
using UnityEngine;
using UnityEngine.UI; // required to use the Text class

public class GameController : MonoBehaviour {

    public Text timeText;

    void Update()
    {
        // ...

        // format to a string with no decimal places
        timeText.text = timeLeft.ToString("0");
    }
```

The game controller can now be given a reference to the Text widget using the inspector, and as the time changes can update the text displayed in the Text widget.

Exercises

Level design

Create an obstacle course of platforms and terrain to be navigated, and 10 targets. Modify the scene, the total time available, and time bonus given by each target to make it easy to shoot the first 5 targets, but difficult to shoot the last 5 targets – aiming for only a few seconds remaining of the total time on completion.

Winning and Losing

Add a second text widget to the HUD canvas. Use this to show whether the player has won (destroyed all the targets) or lost (run out of time)