Space Shooter

Lab 2

In this lab, we'll be looking at using 3D models, making more use of the physics engine and libraries.

Our demo project is a top-down space shooter, along the lines of asteroids.

Part I

Remember

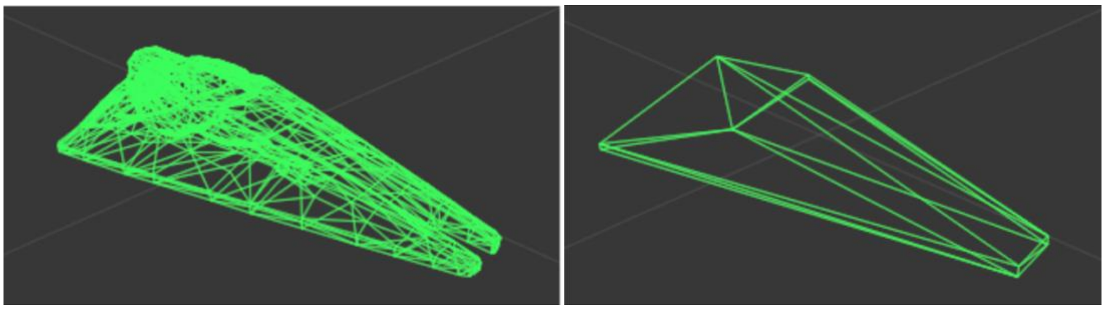
These lab tasks make use of assets that you'll need to download. You can find them on the Moodle.

1.1 Setting up the Project

- Create a new 3D Project and import the supplied assets with this lab (see module website). Use *Assets* → *Import Package* → *Custom Package*
- Save the current scene in the assets directory (or in a sub-folder inside it).
- Change the resolution of the game area: with the *Build Settings* still open, click on Player Settings (or directly from *Edit* → *Project Settings* → *Player*). In the *Inspector View*, set the Resolution's *Default Screen Width* to 600 and *Default Screen Height* to 900. This creates a portrait resolution instead of a landscape one. To do this you'll need to make sure that, "Default is native resolution" is not checked.

1.2 The Player Game Object

- Add the player's model (*Models* → *vehicle_playerShip* in the *Project View*) to the scene. Reset it.
- Change its name in the *Hierarchy View* to "Player".
- Add a *Rigidbody* component to the player. We want it **not** to react to gravity.
- Add a *Mesh Collider* to the player and set it to be *Convex* and a *Trigger*.
    - You'll see that it comes with a *Mesh* property set to "vehicle_playerShip". By turning off the Mesh Renderer component of the player, you'll be able to see the mesh collider in green. Unity checks every triangle of this collider for collisions, which is expensive. It's quite more efficient to use a simpler collider (capsule, box, sphere) or to use a simplified mesh.
    - Luckily, for this model, there is an asset called "vehicle_playerShip_collider" in the *Models* folder. Assign the mesh "player_ship_collider" (child of "vehicle_playerShip_collider") as *Mesh* of the *Mesh Collider* component. This is more efficient, and you can see that this mesh has less triangles.
- Locate the prefab: Prefabs → VFX → Engines → engines_player. Add it as a child of the player. This prefab consists of two particle systems for the player's ship.



1.3 Camera and Lighting

- Set the camera's position to ten units above the player's ship and looking directly at it.
- The camera should be orthographic, as this is an arcade game viewed from the top. Set the size of the camera component to an appropriate value (8 seems to be fair).
- Move the camera along the Z axis so the ship is set in an initial good position (lower half of the screen). Use the *Game View* panel to see how your changes affect the game.
- Set the Background colour of the camera to black. Set also the Environmental Lighting's (*Window → Lighting → Settings*) *source* to Color and set a black colour for it. Delete the Directional Light present in the scene.
- Add a *Directional Light* to the scene (in *Hierarchy View*, *Create → Light → Directional Light*). Give it a name and reset it.
- Modify its transform's rotation so the light hits the player from above and from the right. Choose a colour you like (or just leave it white) and increase its intensity.
- Duplicate the previous light and change its rotation so it hits the player from the front of the player and from the left. Also, change its colour to a different one from the first light (i.e. a light colour), so you can see the difference. Adjust its intensity to your liking.
- Create a third light by duplicating the first one. Set it up so it illuminates the rear of the ship, and from **below** (why do you think we are doing this?), with a dim colour.
- Group all your lights in an empty game object, placed at (0, 100, 0). By doing this, we take all the light gizmos out of the way for an easier manipulation of the scene.

1.4 Adding a Background

- Create a plane and give it a name (it'll hold our background).
- Nothing will be colliding with the background, so the mesh collider component should be removed.
- Add a texture to the plane: "tile_nebula_green_diff" in *Textures*, in the *Project View*. Drag and drop the texture on an empty are in the inspector view, with the plane selected.
- Scale the plane to fill the game view. You should keep the ratio of the original image or the texture will be distorted (how can you check this?).
- Check how different shaders for the plane's material change the way it looks like. Remember that there are a few lights illuminating the background. If we want to avoid the plane being illuminated by these directional lights, you can choose the shader *Unlit → Texture*. This makes the background completely independent from the lighting system, showing the texture's true colour.
- Have a look at how the ship looks like at the moment. Can you see the full model? Why? How would you fix it?

1.5 Moving the Player

- Create a new script for the player's ship. It will control the movement of the ship by changing the rigidbody's position and velocity. The ship should:
  - Move the ship with the cursor keys.
  - Avoid the ship to leave the extents of the game area (it should be always visible). Hint: have a look at the functions of class Mathf.
  - Add a small rotation for lateral movements along the Z axis. The idea is that the ship tilts when it moves from side to side on the X axis. This rotation should be proportional to the horizontal velocity of the ship: no rotation if speed is 0, and a higher rotation for > 0 velocity. Hint: assign the rigidbody's rotation value to the returned Quaternion from the functionQuaternion.Euler ().
  - In which function should you put all this logic?

1.6 Creating Shots

- Create a quad and make sure the quad is visible from the camera. Remove the quad's mesh collider.
- Assign the texture for the bolt (*fx_lazer_orange_diff*). With the bolt selected, drag and drop the texture on the inspector panel.
- The black background of the texture needs to be removed. Change the shader of the material to set it to *Particles → Additive*. This will make the brightest pixels of the texture visible and hide the darkest ones.
- Create a new game object for the physics and behaviour of the shot. Set the quad as a child of this new object.
- Add a rigidbody (set its properties to no respond to gravity) and a collider to this new object (which collider would fit the bolt better?). The collider should be as adjusted to the bolt shape as possible, so play with the values of the collider to adjust it.
- Add a script for the bold behaviour. It should just move the bolt forward at a given speed.
- Save the bolt object as a prefab and remove the bolt from the scene (we will instantiate bolts dynamically).
- Playtest the game and try dragging and dropping bolts into the *Scene View*. You should see bolts spawning in the game and travelling forward.

1.7 Shooting Shots

- Add a new (empty) game object as a child of the player. We'll use the position of this new object as the point in the world where the shots appear. Position this new object in the front of the ship.
- Open the player's script (the one you used to create the movement) and add code to:
  - Instantiate a bolt when the player presses a button of your choice, in the position of the game object created for this. Hint: you need a game object to call the *Instantiate* method, assign the prefab in the editor to a public variable in the script.
  - Add a timer to prevent the player from shooting shots repeatedly, so two consecutive shots must be separated by some minimum time *t*.
- Playtest your game to check that everything works properly. You should see that, pretty quickly, the scene gets fill with bolts that are never destroyed (look at the hierarchy view panel). We'll fix that in the next section.

1.8 Boundary

- Create a new object “Boundary” with a box collider that will act as a trigger. The idea is that a script will destroy the bolts when they collide with this box. You can do it in two different ways:
    - A box is set beyond the limits of the game area, in a way that the bolts will be destroyed when they enter the trigger.
    - A box is set to cover the game area, and the bolts will be destroyed when they leave the trigger.
- In any case, you’ll need a script to actually destroy the game objects. Add the script to the “Boundary” object and destroy the game object using the appropriate function.

1.9 Creating Hazards

- Create a new empty game object for the asteroids. Add the model of the asteroid (Models → *prop_asteroid_01*) to this game object (as a child).
- The asteroid must have a rigidbody (do we want gravity?) and a capsule collider (adjust it to the model).
- Add a new script to the asteroid game object. The objective of this script is to rotate, randomly, the asteroid around its axis. Modify the rigidbody’s angular velocity in the `Start()` function. Hint: have a look at the functions from the `System.Random` class for creating this rotation.
- Playtest the rotation of the asteroid. You should see that the asteroid, after a while, stops rotating. Why does this happen? How could you fix it?
- Add another script that destroys both the bolt and the asteroids when they collide with each other. (Note that you need to have the asteroid’s collider’s *IsTrigger* **not** checked).
- Asteroids should move towards the player. Can you reuse the script you created to move bolts to move the asteroid as well?
- Finally, asteroids should be destroyed when they leave the game area, as bolts are.

1.10 Explosions

- Locate the prefab *explosion_asteroid* in *Prefabs → VFX → Explosions*, in the *Project View*. This is a game object that you can instantiate when an asteroid is destroyed.
- In the script that destroys the asteroid, `Instantiate` an explosion object.
- We need another explosion for the player. Repeat the same process for the player’s explosion, that should happen when the ship collides with an asteroid, using the prefab *explosion_player* in *Prefabs → VFX → Explosions*, in the *Project View*.
- Save the asteroid as a prefab and remove the asteroid from the *Scene View*.

1.11 Game Controller

- A game controller is a script that takes care of general aspects of the game (scoring, spawn hazards, ending the game, etc). This script could be attached to any game object of the scene, or you can have a new one just for this. So, add a new game object to the scene and **tag** it as “GameController”.
- Add a new script to the game controller, named *GameController*.
- Write code to spawn **one asteroid**, at the start the game, in this new script. The asteroid should be spawned at the top of the level: in a random (but within the dimensions of the

playable area) X position, and in a Z position so that the spawning point is outside the camera view.

1.12 Spawning Waves

- Modify the GameController script to spawn multiple asteroids at once.
- Quite possibly, your asteroids collide with each other before reaching the player, which is not very nice. Additionally, when all asteroids are spawned in your (only) wave, no more asteroids appear, and the game becomes quite boring. Spawning asteroids is a *routine* that needs to be executed indefinitely until the game is over. We will use a **coroutine** for that. Hints:
    - A wave is composed by N asteroids. These asteroids should be spawned not at once, but with a little delay between each other. After a wave is finished, another wave should be spawned after a (different) delay.
    - Your coroutine should have a declaration such `IEnumerator SpawnWaves ()` it should be called from `Start()` with the function *StartCoroutine*: `StartCoroutine (SpawnWaves ());`
    - Every time you want to pause the spawning of asteroids (for as long as the two delays explained above specify), you should return `yield return new WaitForSeconds (spawnWait);`
    - Optionally, add a third delay that waits for an initial time until the first wave is instantiated.
- You might have noticed that the explosion game objects stay in the scene even when they have finished. These should be destroyed as well, for efficiency reasons.
    - Create a new script, call it *DestroyByTime* and add it to all the asteroid explosion prefabs.
    - Write the code so the script destroys the explosion game object when the explosion's effect is over (2 seconds is a recommended wait time to destroy these objects).
    - Remember that the changes made should be applied to the prefabs.

Remember

That's the end of the first part! The second part of the lab for week 6 is below.

Part II

In this part we're going to finish off our simple asteroids-style game and add a little polish to it.

2.1 Audio

- In the Audio folder, in the Project View, you'll find several audio clips. All audio clips should be configured to be 2D (remember this setting is in the Audio Source). Assign the following clips to these events in the game:
    - explosion_asteroid: when the explosion of an asteroid starts.
    - explosion_player: when the explosion of the player starts.
    - weapon_player: when the player shoots.
    - music_background: starts at the beginning of the game and loops.
- Adjust the volume and priorities of all audio sources so the background has the highest priority, but explosions are louder than anything else.

2.2 Counting Points and Displaying the Score

- Add an `int score` variable to the GameController script.
- From the script that destroys asteroids when hit by a bolt, do the following:
    - Get a reference to the GameController script.
    - Every time an asteroid is destroyed, increment the score.
- In the editor, create a *Canvas* object and a *UI Text* to update the score of the game. The score should be displayed in the **lower right** corner of the screen. If you don't remember how to do this, check again the instructions of lab 1.

2.3 Ending the Game

- Add another text that appears, at the centre of the screen, only when the player is destroyed. This text should indicate that the game is over.
- Add a third text that appears at the **upper left** corner of the screen, also when the player is destroyed. This text should indicate the player that the game can be restarted by pressing the key 'S'.
- When the texts are ready and working, add the functionality to capture the key press and restart the game. You can reload the application's current scene by making the following call: `SceneManager.LoadScene (SceneManager.GetActiveScene().buildIndex);`. Of course, make sure that reloading the application when pressing 'S' only happens when the game is over.

2.4 A camera cut scene

Implement the following functionality animating the camera at the start of the game. The position the camera has been so far should be the location of the camera during gameplay. However, when the game is started, the camera should be located at the front of the player, looking directly at him. Then, the camera should (gradually) change its orientation and position to its final location. Use the Animation Window for doing this, as shown in the lecture.

You can follow the next steps:

- Create a new camera and give it a position and an orientation that will be used at the beginning of the game.
- Remember to set the projection to Orthographic and give it the same size as in the original camera.
- You probably have a warning saying that you have 2 audio listeners in the scene (it's one per camera). Remove the one from the original camera.
- In the *Project View*, create an Animation object and give it a name. Assign this Animation object as a component of your new camera.
- Open the Animation Window and, with the new camera selected in the *Hierarchy View*, add the transform.position and transform.rotation properties to the Animation.
- Press the record button in the Animation panel and place the time indicator at the end of the animation (by default, this is at 1 : 00). Alternatively, you can do this by going to the last (12th) sample by pressing the forward button (fourth button in the animation window).
- Now, change the position and rotation of the camera so it's placed as in the final gameplay location and orientation. Remember to click again on the record button when you have made these changes. Playtest to see how the camera is animated when you start the game.
- We want the animation to be slower, so go to the Animation Controller, click on the CameraAnimation state, and change the speed to 0.25 (so the animation takes 4 seconds).
- Finally, we don't need the initial camera anymore, so you can remove it from the scene.

2.5 Building the Game

- Open the *Build Settings* window, in the *File* menu. Make sure the *PC* platform is selected and that the current scene is in the list of *Scenes to Build*. If not, make the necessary changes and/or add the current scene to the list by clicking on the *Add Current* button.
- Build the game by clicking on the *Build* button. Select a destination folder (it's advisable to create an exclusive folder for builds), pick a name and click on *Save*.
- You should see that Unity has created an executable file that you should be able to execute and play externally to the Unity editor. Try that now.

Remember

Once complete, get the lecturer or GLA to mark you off on the sheet.