**TEMASEK POLYTECHNIC**

**SCHOOL OF INFORMATICS & IT**

**DIPLOMA IN IMMERSIVE MEDIA & GAME DEVELOPMENT**

**AY2025/2026 APRIL SEMESTER (LEVEL 2)**

**GADV (CGE2C25)**

To see the additional comments and resources, make sure you select **All Markup** in the **Review/Tracking** pane

**Breakout Game**

A screenshot of a video game

Description automatically generated with medium confidence

One good reason for using Unity as an engine for learning how to develop games is that there’s an incredible amount of learning resources online.

The Unity website especially has an excellent collection of video tutorials that cover all aspects of Unity development.

These are mostly in the form of:

* specific topics about the engine itself
* complete end-to-end game development examples
* lengthy live training sessions recorded from Twitch

The Unity website is also where the all-important API documentation and Unity manual lives. YouTube is also a great place to find Unity tutorials. You should get into the habit of using these resources as much as you can – it’s actually a great way to learn.

Today, you are going to follow a tutorial that covers how to create a Breakout game. It’s a good way to learn more about Unity, especially how a basic arcade style game is put together.

**Task 1**

Follow the video tutorial here:

<https://tinyurl.com/3ytd5dmx>

Implement the game, which you must demo to your tutor in class.

If you have any questions about any part of the videos, ask your tutor for help! If enough people have the same question, your tutor will give a brief explanation to the whole class.

These notes will help you to understand some of the less obvious parts of the video.

|  |  |
| --- | --- |
| Time | Note |
| 0:12 | The camera can have different backgrounds. Here, the background is changed from the default skybox to a solid colour. The Background colour picker is usually near the top of the Camera component, but this may vary depending on your Unity version. |
| 0:20 | The ball, paddle and wall objects are created as empty GameObjects first. The required components for the paddle are then added:   * SpriteRenderer: to display an image (render a sprite) * Rigidbody2D: to use Unity 2D physics * BoxCollider2D: a box collider for 2D objects (2D and 3D physics components will ignore each other) |
| 0:40 | A sprite is dragged to the Sprite slot of the SpriteRenderer component. An image can be imported into Unity as a Texture (to put on a 3D object) or as a Sprite (which will be displayed as a 2D image in the game). |
| 1:00 | When an object is created, its collider will automatically have the same dimensions. Here, the size of the sprite was changed, so the collider is removed and then added again to set it to the correct size. Collider resizing can be done manually, but this way is a bit faster. |
| 1:08 | To manually edit a collider, click the **Edit Collider** button and then drag the small boxes as shown in the video. In this case, the collider edges are moved to match the sprite edges. |
| 1:15 | Some of the PaddleController script has been written already. It simply moves the paddle according to the player input (left or right arrow keys). A better way to get input might be to use Input.GetAxis(“Horizontal”), which would use the more abstract features of the input system, but this is often a matter of preference.  Note that the scripts are placed in a namespace. This is a way to separate scripts into different “collections”. One advantage is that classes can have the same names without causing a compiler error (the compiler knows that each class is in a different namespace). |
| 1:15 | Note that the Rigidbody2D is stored as a variable, **\_myRb**. The Rigidbody2D component is then obtained using **GetComponent<Rigidbody2D>()** in the **Start()** function. It’s better to do this, since GetComponent is a relatively expensive function in terms of CPU time. Caching (storing) the value in a variable means that GetComponent is only called once when the game object is initialised. |
| 1:15 | Vector2.left is a vector pointing in the negative X direction with a magnitude of 1. Vector2.right points in the positive right direction with a magnitude of 1. This is stored in **direction**, whichwill be used later to add a force to move the paddle left or right. |
| 1:35 | Remember that physics related code should be put in FixedUpdate(). This is so that the physics engine can update all the physics objects at a regular timestep. |
| 1:45 | The **\_direction** vector has a magnitude of 1. A force of 1 is very weak, and probably won’t move the paddle much (this depends on the mass we give it). To increase the speed of the paddle, we can multiply the vector by a float value, which is called **speed** here. |
| 1:50 | The **speed** variable is made public so that it’s value can be set in the Inspector. Note: if for you want a variable to be public but not visible in the Inspector (to stop a level designer messing with it, say), you can put [HideInInspector] in front of its declaration, e.g.,  [HideInInspector] public float speed = 10f;  It’s important to remember that whatever value you initialize a variable to, that value will be overridden by the value in the Inspector when the game starts. |
| 2:00 | The Rigidbody2D’s Gravity Scale property is set to 0, so that the paddle doesn’t fall due to gravity. Adding some Linear Drag makes the movement more realistic and less “slippery”. Changing the Mass will affect the paddle’s speed when a force is applied. The Rigidbody2D is constrained along the Y axis (so it doesn’t move up or down), and by rotation around the Z axis (to stop it rotating when hit by the ball). |
| 2:30 | Games are not meant to be scientifically accurate simulations. So, for game physics, the usual rule is “If it looks good, then fine.” Also, game physics are often deliberately not realistic. For the paddle, the various physics properties are tweaked until the paddle movement looks good. |
| 3:00 | The ball sprite is added to the Ball empty game object, just as for the paddle. |
| 3:25 | A Physic Material specifies the physics properties of an object when interacting with other physics objects. Here, a Physic Material called **Bouncy Ball** has been created to make the ball bouncier. This is dragged into the Rigidbody2D’s **Material** slot. |
| 3:55 | This script is like the paddle script, in that it is concerned with moving the ball.  Line 14 uses Unity’s **Invoke()** method, which is inherited from MonoBehaviour. Invoke() simply calls a method after a given time delay. In this case, the **SetRandomTrajectory** method is called after a 1 second delay. Note Unity’s **nameof** function, which returns the name of a method as a string. |
| 4:00 | Remember that **Awake()** and **Start()** are inherited from **MonoBehaviour** and are called automatically as part of the MonoBehaviour lifecycle. **Awake()** is called before **Start().** |
| 5:00 | Each wall is created as an empty game object. In this case, each wall is just a boundary for the ball to bounce off, so we don’t need to display any sprite. Only a BoxCollider2D is added, and then resized manually to fit the edges of the screen.  Note that all the wall objects are parented to the **Walls** empty game object. This is a good way to organize game objects in your Hierarchy window. |
| 6:30 | The obstacles (bricks) are created in the same way as the paddles. |
| 6:45 | An obstacle is not moved by physics, so doesn’t need a Rigidbody2D. All it needs is a BoxCollider2D to make the ball collide with it. NOTE: for a collision to occur, at least one of the colliding objects must have a Rigidbody2D, in this case the ball. |
| 7:00 | Since we will create many obstacles, it’s best to make an Obstacle prefab. This means that we can make all the changes we want to the prefab, and then instantiate as many of these as we like. There’s no need to change the properties of each prefab separately. And if we change any of the prefab properties, these changes will immediately be applied to all the prefab instances (see 7:20). Note that it’s good practice to put prefabs in a separate **Prefabs** folder. |
| 8:10 | The ball’s Z scale is not changed, since for a 2D game the Z axis points out of the screen, so any changes won’t be visible (also, as a 2D object, it doesn’t have any depth, even though it does have a Z scale value of 1). |
| 8:45 | Note the **OnCollisionEnter2D** method. This is inherited from MonoBehaviour, so what we are doing is overriding this method to put our own collision code. This method is called whenever a brick collides with another object with a collider component. The **Collider2D** argument, **other**, has information about the colliding object.  The ball object has been given a tag, “Ball” (see 9:05). To get the GameObject that **other** refers to simply access its gameObject property (line 6). We can then check the game object’s tag. If the tag isn’t “Ball”, do nothing and return immediately, otherwise destroy the brick using Unity’s **Destroy()** method. |

**Task 2**

Following a game tutorial is a great way to learn, but it can give you a false sense of security, where you think it’s *your* game (“Look at my breakout game!”) but all the hard work was really done by whoever wrote the tutorial.

To make sure you do understand, and also to have more of a sense of achievement, you should try to add some enhancement(s) to the game. That’s what you’ll do in this session. You can work in pairs, as long as you each contribute to the discussion, research, and implementation.

IMPORTANT: Don’t worry if you can’t complete it. At this stage, the attempt to do something by yourself is more important than the end result. Also, if you really find it too difficult, choose something a bit simpler first.

The main focus in GADV is actually to try to get you used to *problem-solving*. Once you start making your own (hopefully unique) games, you may have to solve problems that no one has attempted before!

* Look at the list of enhancements below and choose one to implement. No more than TWO students can do the same enhancement (or two pairs).
* The implementation is your homework for this week – you should be ready to present your work next week. If you can’t complete it, at least discuss what you did, why you found it difficult, and what you think you need to learn before you can solve the problem.

**Enhancements**

Here’s a list of possible enhancements to the basic Breakout game. If you can think of your own enhancement, do that - but don’t be too ambitious just yet.

We haven’t covered many of the concepts needed to implement these, but the best way to learn Unity is simply to dive into a problem and try to figure it out with the help of documentation, tutorials, online forums, discussion with classmates, and eventually some help from your tutor.

|  |  |
| --- | --- |
|  | **Description** |
| 1 | Have multiple scenes with different arrangements of bricks. You must figure out how to load a different scene into your game. |
| 2 | Modify the game so that the bricks oscillate left and right during the game. |
| 3 | Modify the game so that removed bricks will reappear in their original position after a random amount of time within a specific time range that you may set. |
| 4 | Implement a time-limit to the game, with a countdown timer displayed. |
| 5 | Implement a set number of lives for the player, which the player can see. |
| 6 | Add sound effects to the game. |
| 7 | Add objects to the game which affect the movement and action of the ball, e.g.:   * floating “rocks” that change the ball’s direction, but which do not disappear when hit * a barrier which moves left and right – if the ball hits the barrier, the ball will bounce off it, but the barrier will not disappear * floating objects which cause the ball to “teleport” to a random location. (Can you figure out how to make it teleport to an *empty* location?!) |
| 8 | Add some form of visual effects to the game (e.g., a particle system). You should study the Unity documentation to find out what visual effects are possible. |
| 9 | When a brick is hit, it does not immediately disappear, but fades away over a given length of time. Once a brick is hit, it *does not* interact with the ball again, even if the brick has not completely disappeared. |
| 10 | Imagine you want to review the movement of the ball for a game to make sure that the ball bounces correctly. A line is drawn to trace the ball’s movement. Technically this is called an object trail. An example is given in the image below: |
| 11 | Modify the game so that at random intervals an extra ball is added, e.g., after 10 seconds, a second ball is added, after 25 seconds, a third ball is added, etc. |
| 12 | Implement a booster object. When the ball enters the booster object, the ball’s speed is increased by a specific amount in the direction that it is moving. The ball’s speed then slows down over a specified length of time to the original speed. The booster only works when the ball is moving up. You should look at how to implement triggers in Unity. |
| 13 | Each time the game starts, the speed and angle of movement of the ball will be random. However, the vertical direction will always be UP. |
| 14 | Each row of bricks must be a different colour – and every time the game starts, each row of bricks is set to a random colour. |
| 15 | As the game plays, the bricks move slowly down towards the bottom, like space invaders. |