2019

Monday 19 December 2022 02:08

Question 1

A 3D tower defence game made in Unity has the following rules:

1. Players place towers by using the mouse to select a location on the map.
2. Towers become active when a creep comes in range.
3. When a tower becomes active it will turn to face the creep and continue targeting the creep so long as it stays in range.
4. Towers can fire 5 bullets per second.
5. Bullets disappear after 5 seconds if they don't hit anything
6. There are three possible types of creeps and each has an equal probability of being spawned.
7. Creeps follow a path to get to the players base.
8. When a creep is hit with a bullet, it explodes and after a few seconds, sinks into the ground and gets removed from the scene.

Taking each of the rules above, how would you program them in Unity?

(8 x 5 marks)

Here is one possible way to program these rules in Unity:

1. Players place towers by using the mouse to select a location on the map: To implement this, you can use Unity's built-in Input system to detect mouse clicks on the map. When a mouse click is detected, you can check the location of the click and instantiate a tower prefab at that location.

2. Towers become active when a creep comes in range: To implement this, you can use Unity's built-in Physics system to detect when a creep enters the tower's trigger collider. When this happens, you can set a flag on the tower indicating that it is now active.

3. When a tower becomes active it will turn to face the creep and continue targeting the creep so long as it stays in range: To implement this, you can use the tower's transform component to rotate the tower towards the creep's position. You can then use a coroutine or update function to continuously check whether the creep is still in range and update the tower's rotation accordingly.

4. Towers can fire 5 bullets per second: To implement this, you can use a timer or a counter to keep track of the time elapsed since the last bullet was fired. When the timer reaches a certain threshold (e.g. 0.2 seconds for 5 bullets per second), you can instantiate a bullet prefab and reset the timer.

5. Bullets disappear after 5 seconds if they don't hit anything: To implement this, you can use a timer or a countdown function to keep track of the time elapsed since the bullet was instantiated. When the timer reaches 5 seconds, you can destroy the bullet game object.

6. There are three possible types of creeps and each has an equal probability of being spawned: To implement this, you can use Unity's built-in Random class to generate a random number between 1 and 3. Based on the value of the random number, you can instantiate one of the three creep prefabs.

7. Creeps follow a path to get to the player's base: To implement this, you can create a series of waypoints along the path that the creeps should follow. You can then use Unity's built-in NavMesh system to guide the creeps from one waypoint to the next.

8. When a creep is hit with a bullet, it explodes and after a few seconds, sinks into the

ground and gets removed from the scene: To implement this, you can use Unity's built-in Particle System to create an explosion effect when the bullet hits the creep. You can then use a timer or a coroutine to wait a few seconds before sinking the creep into the ground and destroying it.

Question 2

(a) When generating a mesh procedurally, a programmer may generate arrays of *vertices, normals, colors, triangles,* and *uv.* Identify the purpose and data type of each of the italicised terms.

(15 marks)

(b) A terrain mesh contains a grid of quads of size 50 x 50. How many vertices are in this mesh assuming no vertices are shared between triangles?

(5 marks)

(c) Figure 1 is an extract from a game component that generates a terrain mesh. Explain the main features of this code.

(10 marks)

```
Vector? bottomLeft = new Vector3(-samples.x / 2, 0, -samples.y / 2);
coll = AddComponent<MeshCollider>();
   int vertex = 0;
    for (int y = 0; y < samples.y; y++)
    <
          for (int x = 0; x < samples.x; x++)
          {
                Vectors sliceBottomLeft = bottomLeft + new Vector3(x, 0, y);
                Vectors sliceTopLeft = bottomLeft + new Vectors^, 0^ y + 1);
                Vectors sliceTopRight = bottomLeft + new VectorS(x + I, 0, y + 1);
                Vectors sliceBottomRight = bottomLeft + new Vector3(x + 1, 0, y);

                sliceBottomLeft.y += SampleCell(x, y) * amplitude;
                sliceTopLeft.y += SampleCell(x, y + 1) * amplitude; sliceTopRight.y += SampleCell(x + 1, y +
                1) * amplitude; sliceBottomRight.y += SampleCell(x + I, y) * amplitude;

                 int startvertex = vertex;
                 gm.initialVertices[vertex++] = sliceBottomLeft;
               gm.initialVertices[vertex++] = sliceTopLeft;
               gm.initialVertices[vertex++] = sliceTopRight;
               gm.initialVertices[vertex++] = sliceTopRight; gm.initialVertices[vertex++] = sliceBottomRight;
              gm.initialvertices[vertex++] = sliceBottomLeft;

               for (int i = 0; i < 6; i++)
               {
                     gm.meshUv[startVertex + i] = new Vector2(x / samples.x, y / samples.y);
                     gm.meshTriangles[startVertex + i] = startvertex + i;
               }
         }
  }

  mesh. vertices = gm.initialvertices;
  mesh.uv = gm.meshUv;
  mesh.triangles = gm.meshTriangles;
  mesh.RecalculateNormals();

  coll.sharedMesh = null;
  coll.sharedMesh = mesh;
```

Figure 1

(a)

Vertices: An array of Vector3 values representing the 3D coordinates of the points on the mesh. This array is used to specify the positions of the vertices in 3D space. The vertices are connected to form the edges and faces of the mesh. The data type of this array is Vector3[], which is an array of Vector3 values.

Normals: An array of Vector3 values representing the surface normals of the mesh. The normal of a vertex is a vector that is perpendicular to the surface at that point.

Normals are used to determine the direction that a surface is facing, which is important for rendering the mesh correctly and for applying lighting to the surface. The data type of this array is Vector3[], which is an array of Vector3 values.

Colors: An array of Color values representing the colors of the vertices on the mesh. The colors are used to specify the color of each vertex on the mesh, which can be used for various purposes such as coloring different parts of the mesh differently or applying a color gradient to the surface. The data type of this array is Color[], which is an array of Color values.

Triangles: An array of integers representing the indices of the vertices that make up the triangles of the mesh. A triangle is a 3-sided polygon that is formed by connecting three vertices on the mesh. The triangles array specifies the indices of the vertices that make up each triangle in the mesh. The data type of this array is int[], which is an array of integers.

UV: An array of Vector2 values representing the 2D texture coordinates of the vertices on the mesh. The UV coordinates are used to map a 2D texture onto the surface of the mesh. They specify the location on the texture that should be applied to each vertex on the mesh. The data type of this array is Vector2[], which is an array of Vector2 values.

(b)
There are 50 x 50 = 2500 quads in the mesh, and each quad has 4 vertices. This means that there are a total of 2500 x 4 = 10000 vertices in the mesh.
The assumption here is that no vertices are shared between triangles, which means that each vertex is used by a single triangle and is not shared with any other triangles. This means that the total number of vertices in the mesh is equal to the total number of triangles in the mesh.
If vertices were shared between triangles (i.e. if the mesh used indexed vertices), then the total number of vertices in the mesh would be less than the total number of triangles, because each triangle would use 3 vertices from the vertex array, but some of these vertices may be shared between multiple triangles.

(c)
This code generates a mesh procedurally by creating a grid of quads and sampling the height of each vertex using the "SampleCell" function.

First, the bottom left corner of the grid is defined as a Vector3 value called "bottomLeft". Then, a MeshCollider component called "coll" is added to the game object.

Next, a loop iterates over the x and y values of the samples, which represent the rows and columns of the grid. For each x and y value, the vertices of the quad at that position are created by adding an offset to the bottom left corner of the grid. The y-coordinate of each vertex is then modified based on the value returned by the "SampleCell" function, which is multiplied by the "amplitude" value.

The UV coordinates and triangles of the mesh are also generated in this loop. The UV coordinates are set to the normalized x and y values of the current quad, and the triangles are defined using the indices of the vertices that make up the quad.

After the loop completes, the mesh is updated with the new vertices, UV coordinates, and triangles, and the mesh collider component is updated to use the new mesh. The "RecalculateNormals" function is also called to recalculate the normals of the mesh based on the new vertices.

Question 3

(a) Figure 2 shows an extract from a generative physics system that creates the caterpillar animat given in Figure 3.

```
 void Awake()
{
      float depth = size * 0.05f;
      Vectors start = - Vectors.forward * bodySegments * depth * 2;
      GameObject previous = null;
      for (int i = 0; i < bodySegments; i++)
      {
            float mass = 1.0f;
            GameObject segment =
GameObject.GreatePrimitive(PrimitiveType.Cube);
            Rigidbody rb = segment.AddComponent<Rigidbody>();
            rb.useGravity = gravity;
            rb.mass = mass;
            segment.name = "segment " + i;
            Vectors pos = start + (Vectors.forward * depth * 4 * i);
            segment. transform. position = transform.TransformPoint(pos);
            segment.transform.rotation = transform.rotation;
            segment. transform. parent = this.transform;
            segment .transform, localscale = new Vector3(size., size, depth);
            segment.      GetComponent<Renderer>().shadowCastingMode      =      UnityEngine.
Rendering.ShadowCastingMode. Off;
            segment.GetComponent<Renderer>().receiveShadows = false;

               segment. GetComponent<Renderer>(). material. color =
  Color.HSVToRGB(i / (float)bodySegments,                1);

            if (previous != null)
            {
                  j. autoConfigureConnectedAnchor = false;
                  j.anchor = new Vector3(0j -2f);
                  j.connectedAnchor = new Vector3(0, 0, 2f); j.axis = Vectors. right;
                  j.useSpring = true;
                  Jointspring js = j.spring;
                  js.spring = spring;
                  js.damper = damper;
                  j.spring = js;                    }
            previous = segment;
      }
}
```

Figure 2

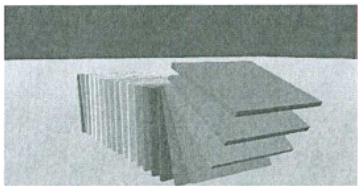

Figure 3

Answer these questions about the code:

(a) How are the position, rotation, anchor points and scale of each segment calculated? Include a diagram in your answer.

m

(10 marks)

(b) How are the segments constrained to move relative to one another?

(10 marks)

(c) How is the colour of each segment determined?

(3 marks)

(d) What should the hierarchy look like after the Awake method has been called?

(2 marks)

(e) How would you procedurally animate the caterpillar so that torque is applied to to each segment in sequence?

(5 marks)

(a) The position, rotation, and scale of each segment are calculated as follows:

Position: The position of each segment is calculated by adding an offset to the starting position, which is defined as "start". The starting position is calculated as the negative of the forward direction multiplied by the number of body segments

multiplied by the depth of each segment multiplied by 2. The depth of each segment is defined as "size * 0.05f". The offset is calculated as the forward direction multiplied by the depth of each segment multiplied by 4 multiplied by the current iteration of the loop (i.e. the index of the segment). The final position of the segment is then transformed into world space using the transform of the parent object.

Rotation: The rotation of each segment is set to the rotation of the parent object.
Scale: The scale of each segment is set to a new Vector3 value with the x and y values set to "size" and the z value set to "depth".

Anchor points: The anchor points of the joint between each segment are calculated as follows:

- The anchor point of the current segment is set to a new Vector3 value with the x and y values set to 0 and the z value set to -2f.

- The connected anchor point of the previous segment (if it exists) is set to a new Vector3 value with the x and y values set to 0 and the z value set to 2f.

In other words, the position of each segment is calculated by adding an offset to the starting position, which is defined as the negative of the forward direction multiplied by the number of body segments multiplied by the depth of each segment multiplied by 2. The offset is calculated as the forward direction multiplied by the depth of each segment multiplied by 4 multiplied by the current iteration of the loop. The rotation of each segment is set to the rotation of the parent object, and the scale of each segment is set to a new Vector3 value with the x and y values set to "size" and the z value set to "depth". The anchor point of the current segment is set to a new Vector3 value with the x and y values set to 0 and the z value set to -2f, and the connected anchor point of the previous segment (if it exists) is set to a new Vector3 value with the x and y values set to 0 and the z value set to 2f.

This means that the position of each segment is calculated by adding an offset to the starting position, which is defined as the negative of the forward direction multiplied by the number of body segments multiplied by the depth of each segment multiplied by 2. The offset is calculated as the forward direction multiplied by the depth of each segment multiplied by 4 multiplied by the current iteration of the loop. The rotation of each segment is set to the rotation of the parent object, and the scale of each segment is set to a new Vector3 value with the x and y values set to "size" and the z value set to "depth". The anchor point of the current segment is set to a point 2 units behind the center of the segment, and the connected anchor point of the previous segment (if it exists) is set to a point 2 units in front of the center of the previous segment.

(b) How are the segments constrained to move relative to one another?

(10 marks)

The segments are constrained to move relative to one another using a hinge joint component.

A hinge joint component is added to each segment except the first one (which doesn't have a previous segment to be connected to). The anchor point of the current segment and the connected anchor point of the previous segment are set to the positions described in the previous answer. The axis of the joint is set to the right direction, and the useSpring property is set to true.

The spring property of the hinge joint is then configured using a JointSpring object. The spring and damper properties of the JointSpring object are set to the "spring" and "damper" values defined in the code, respectively.

This means that each segment is connected to the previous segment using a hinge joint that allows the segments to rotate around the joint axis (which is set to the right direction) and is constrained by a spring force with the specified spring constant and damping coefficient. This causes the segments to move relative to one another in a way that simulates a flexible body, such as a caterpillar.

(c) How is the colour of each segment determined?

(3 marks)

(d) What should the hierarchy look like after the Awake method has been called?

(2 marks)

(e) How would you procedurally animate the caterpillar so that torque is applied to to each segment in sequence?

(5 marks)

(c) The color of each segment is determined by converting the hue, saturation, and value of a color to the red, green, and blue channels using the "HSVToRGB" function. The hue value is calculated as the current iteration of the loop divided by the number of body segments. The saturation and value values are both set to 1. This means that each segment is given a different hue value based on its position in the sequence, and the saturation and value are both set to maximum.

(d) The hierarchy should look like a parent object with a number of child objects representing the segments of the caterpillar. The parent object should have a "Caterpillar" component attached to it, and each child object should have a "Cube" component attached to it. The name of each child object should be "segment i", where i is the index of the segment.

(e) To procedurally animate the caterpillar so that torque is applied to each segment in sequence, you could use a loop that iterates over each segment and applies a torque to it.

For example:

```
for (int i = 0; i < bodySegments; i++)
{
  GameObject segment = transform.Find("segment " + i).gameObject;
  Rigidbody rb = segment.GetComponent<Rigidbody>();
  rb.AddTorque(Vector3.up * torque, ForceMode.Impulse);
}
```

Question 4

(a) What are the main features of the C# Job System?

The C# Job System in Unity is a system for writing multi-threaded code in a way that is safe and easy to use. It allows you to write code that can be run in parallel on multiple threads, making it possible to take advantage of multiple CPU cores and improve the performance of your game or application.

Some of the main features of the C# Job System in Unity include:

1. Job scheduling: The C# Job System provides a way to schedule jobs to run on worker threads. You can specify which jobs should run in parallel and which jobs should run sequentially, and the system will take care of assigning the jobs to available worker threads.

2. Data dependencies: The C# Job System allows you to specify data dependencies between jobs, which ensures that the jobs are executed in the correct order. For

example, you can specify that a job should only start running after another job has completed.

3. Memory safety: The C# Job System provides a way to access and manipulate data in a thread-safe way, ensuring that data is not modified concurrently by multiple threads. This is important to prevent race conditions and other types of data corruption.

4. Performance: The C# Job System can significantly improve the performance of your game or application by allowing you to parallelize tasks that would otherwise be run on a single thread. This can be especially useful for tasks that are computationally intensive or that involve a lot of data processing.

5. Ease of use: The C# Job System is designed to be easy to use, with a simple API and clear documentation. It is also integrated with the rest of the Unity engine, making it easy to use with other Unity features and systems.

(b) Figure 4 shows an extract from a procedural animation system that implements a harmonic motion. How would you convert this code to use the C# Job System? In your solution include a description of:

i .What new classes/structs that you would need to create. (5 marks)

ii .What fields and their types would need to be on these new classes/structs. (10 marks)

iii .What methods you would need to create on the new classes/structs (10 marks)

```
public class Sway : MonoBehaviour {
    public float angle = 20.0f;
    public float frequency;
    public float theta;
    public Vectors axis = Vectors. zero;
    // Use this for initialization
    void Start () {
        if (axis == Vectors.zero)
        {
            axis = Random.insideUnitSphere;
            axis.y = 0;
            axis.Normalize();
        }
    } '
    void Update () {
        transform.localRotation = Quaternion.AngleAxis(
            Mathf.Sin(theta) * angle, axis);
        theta += frequency * Time.deltaTime * Mathf.PI * 2.0f;
    }
}
```

Figure 4

(i) New classes/structs:
To convert the code to use the C# Job System, you would need to create new classes or structs to represent the job and the data that the job will operate on.
The struct that represents the data for the job would contain the necessary information that the job needs to operate on. For example, in the case of the procedural animation system, the data struct might contain fields for the angle, frequency, theta, and axis values that are needed to perform the harmonic motion animation.
The class that represents the job would contain the logic for performing the task that the job is responsible for. For example, in the case of the procedural animation system, the job class might contain a method that applies the harmonic motion animation to a transform using the angle, frequency, theta, and axis values that are provided in the data struct.
By separating the data and the logic into separate classes or structs, you can make it easier to write and manage your code, and you can also take advantage of the C#

Job System's ability to parallelize tasks by scheduling multiple instances of the same job with different data.

(ii) Fields and their types:
The fields of the struct that represents the data for the job would need to be of the same types as the fields in the "Sway" class.

For example:
- A "float" field for "angle"
- A "float" field for "frequency"
- A "float" field for "theta"
- A "Vector3" field for "axis"

The fields of the class that represents the job would need to be of the same types as the fields in the "Sway" class, plus any additional fields that are needed by the job.

For example:
- A "float" field for "angle"
- A "float" field for "frequency"
- A "float" field for "theta"
- A "Vector3" field for "axis"
- A "Transform" field for "transform"

The "transform" field would be used to store a reference to the transform of the object that the job is operating on. This would be necessary because the job will be running on a worker thread, and it will not have access to the "transform" field of the "Sway" component.

By storing the data and the logic in separate classes or structs, you can make it easier to manage the fields and data that are used by the job. You can also take advantage of the C# Job System's memory safety features, which ensure that data is accessed and modified in a thread-safe way.

(iii) Methods:

The class that represents the job would need to have the following methods:
- A constructor that takes a "SwayData" struct as an argument and assigns the values of the struct to the fields of the class. This constructor would be used to initialize an instance of the "SwayJob" class with the data that it needs to operate on.

- A "Execute" method that performs the harmonic motion animation. This method would be marked with the "Execute" attribute to indicate that it is a job method. The "Execute" method would use the "angle", "frequency", "theta", and "axis" fields to apply the harmonic motion animation to the "transform" field.
Here is an example of what the "Execute" method might look like:

```
public void Execute()
{
   transform.localRotation = Quaternion.AngleAxis(
      Mathf.Sin(theta) * angle, axis
   );
   theta += frequency * Time.deltaTime * Mathf.PI * 2.0f;
}
```

By creating an "Execute" method on the job class, you can specify the logic that the job should execute when it is run on a worker thread. The "Execute" method is

where you would put the code that was previously in the "Update" method of the "Sway" class.

By using the C# Job System, you can write code that is easy to understand and maintain, and you can also take advantage of the system's ability to parallelize tasks and improve the performance of your game or application.