2020

Monday 19 December 2022 02:08

Question 1

A spawner system for creatures in a virtual world programmed in C# for Unity has the following features:

1. There is a list of possible creature types and these are spawned in order.
2. Creatures are spawned inside the players field of view, a random distance from the player between a minimum and maximum range.
3. Creatures spawns on top of the environment.
4. Creatures that are a certain distance behind the player get suspended.
5. When a new creature is needed, a previously suspended creature can be used instead of instantiating a new one.
6. Creatures fade in when they are created.
7. Creatures are spawned and suspended at a programmer specified rate.
8. The spawner is a singleton.

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

(8 x 5 marks)

Here is how you could program the features of the spawner system in Unity using C#:

1. List of possible creature types: You can create a list of prefabs for each type of creature. You can then use a loop to iterate through the list and instantiate the prefabs in order.

2. Spawn inside player's field of view: You can use the Camera.main object to get the player's field of view. You can then use the camera's Transform component to get the player's position and rotation, and use these values to calculate a random spawn position within the field of view.

3. Spawn on top of environment: To spawn the creatures on top of the environment, you can use a raycast to find the position of the ground below the spawn position. You can then set the creature's position to this ground position.

4. Suspend creatures behind player: You can use the player's Transform component to get their position, and then use this position to calculate the distance between the player and each creature. If a creature is a certain distance behind the player, you can set its active state to false to suspend it.

5. Use suspended creatures for new ones: When you need to spawn a new creature, you can check if there are any suspended creatures available. If there are, you can set the active state of one of these creatures to true and use it instead of instantiating a new one.

6. Fade in when created: To fade in the creatures when they are created, you can use an animation or tweening library to gradually increase their transparency over a certain period of time.

7. Spawn and suspend at specified rate: You can use a timer or a coroutine to control the rate at which creatures are spawned and suspended. You can set the timer or coroutine to run at the desired rate and use it to trigger the spawning and suspending of the creatures.

8. Singleton spawner: To create a singleton spawner, you can use the Singleton pattern. This involves creating a static instance of the spawner class, and using a private constructor to prevent multiple instances of the spawner from being created. You can then provide a public static method to access the instance of the spawner.

Question 2

(a) Discuss the relationship between the quantities of *force*, *velocity*, *acceleration*, *position*, *distance*, *time* and *mass* in relation to 3D computer games. In your answer include:

i. Units of measurement and representations for these quantities. (5 marks)
ii. Equations that describe the relationships. (5 marks)
iii. A description of how to update the state of a Newtonian physics particle with respect to time in a 3D computer game. (5 marks)

i. Units of measurement and representations for these quantities:

- Force: In physics, force is usually measured in units of newtons (N). In a 3D computer game, force can be represented as a 3D vector, with each component of the vector representing the force applied in the x, y, and z directions.
- Velocity: Velocity is a measure of the speed and direction of an object's movement. It is usually measured in meters per second (m/s). In a 3D computer game, velocity can also be represented as a 3D vector, with each component representing the velocity in the x, y, and z directions.
- Acceleration: Acceleration is a measure of how quickly an object's velocity changes. It is usually measured in meters per second squared (m/s^2). In a 3D computer game, acceleration can also be represented as a 3D vector.
- Position: Position is a measure of the location of an object in 3D space. It is usually measured in meters. In a 3D computer game, position can be represented as a 3D vector, with each component representing the x, y, and z coordinates of the object.
- Distance: Distance is a measure of the separation between two objects. It is usually measured in meters. In a 3D computer game, distance can be calculated as the magnitude of the difference between the two objects' positions.
- Time: Time is a measure of the duration of an event. It is usually measured in seconds. In a 3D computer game, time can be represented as a scalar value.
- Mass: Mass is a measure of the amount of matter in an object. It is usually measured in kilograms. In a 3D computer game, mass can be represented as a scalar value.

ii. Equations that describe the relationships:

- Force: The relationship between force and mass is described by Newton's second law of motion, which states that the force applied to an object is equal to its mass multiplied by its acceleration: F = ma.
- Velocity: The relationship between velocity and acceleration is described by the equation v = at, where v is velocity, a is acceleration, and t is time.
- Acceleration: The relationship between acceleration and velocity is described by the equation a = (v_f - v_i)/t, where a is acceleration, v_f is the final velocity, v_i is the initial velocity, and t is time.
- Position: The relationship between position and velocity is described by the equation x = x_0 + v_0t + (1/2)at^2, where x is position, x_0 is the initial position, v_0 is the initial velocity, a is acceleration, and t is time.
- Distance: The distance traveled by an object can be calculated as the magnitude of the displacement vector, which is the difference between the final position and the initial position.
- Time: The time it takes for an object to travel a certain distance can be calculated using the equation t = d/v, where t is time, d is distance, and v is velocity.

iii. Updating the state of a Newtonian physics particle:

To update the state of a Newtonian physics particle in a 3D computer game, you can use the equations described above to calculate the particle's new position, velocity, and acceleration based on its current state and the forces acting on it. This process

can be repeated at regular intervals, usually at a rate of 60 times per second, to simulate the movement of the particle over time.

Here is an example of how you could update the state of a particle in a 3D computer game:

1. Determine the forces acting on the particle: In order to update the state of the particle, you need to know the forces that are acting on it. These forces can include gravitational force, friction, and any external forces that may be applied to the particle, such as those resulting from collisions with other objects.

2. Calculate the acceleration: Once you know the forces acting on the particle, you can use Newton's second law of motion (F = ma) to calculate the acceleration of the particle.

3. Update the velocity: Using the equation v = at, you can calculate the change in velocity of the particle based on its acceleration and the elapsed time since the last update. You can then add this change in velocity to the particle's current velocity to get the new velocity.

4. Update the position: Using the equation x = x_0 + v_0t + (1/2)at^2, you can calculate the change in position of the particle based on its initial position, initial velocity, acceleration, and the elapsed time since the last update. You can then add this change in position to the particle's current position to get the new position.

5. Repeat the process: To continue simulating the movement of the particle over time, you can repeat these steps at regular intervals.

(b) A *gravity gun* in 3D games allows the player to grab an object and hold it at a point in front of the camera. Explain in detail how you would implement a gravity gun effect in a Unity project.

(15 marks)

To implement a gravity gun effect in a Unity project, you can follow these steps:

1. Create the gravity gun object: First, you need to create the object that will represent the gravity gun in the game world. This can be done by creating a 3D model of the gun and importing it into Unity.

2. Attach a script to the gravity gun object: Next, you need to attach a script to the gravity gun object that will handle the logic of the gravity gun effect. This script will need to be able to detect when the player is aiming the gun at an object and when the player is firing the gun.

3. Implement object detection: To detect when the player is aiming the gun at an object, you can use a raycast to shoot a line from the gun's position in the direction that the gun is pointing. If the raycast hits an object, you can store a reference to that object and highlight it to indicate that it can be grabbed by the gravity gun.

4. Implement object grabbing: To implement object grabbing, you can use the physics system in Unity to apply a force to the grabbed object that causes it to move towards the gun. You can also use the physics system to apply a force to the grabbed object to keep it at a fixed distance from the gun.

5. Implement object releasing: When the player releases the object, you can remove the forces applied to the object and allow it to behave according to the laws of

physics.

6. Implement other features: You can also implement other features of the gravity gun, such as the ability to adjust the strength of the force applied to the grabbed object or the ability to rotate the grabbed object.

7. Test and debug: Finally, you can test and debug the gravity gun effect to ensure that it is working correctly and that it behaves as expected. This may involve adjusting the values of the forces applied to the grabbed object or adding additional code to handle edge cases or errors.

Question 3

(a) In relation to digital audio, explain the following terms: *sample rate, resolution, frame size, spectrum, bin width.*

(10 marks)

- Sample rate: The sample rate of a digital audio signal is the number of samples taken per second to represent the audio waveform. The sample rate is usually measured in kilohertz (kHz). A higher sample rate results in higher quality audio, but also requires more storage space and processing power.

- Resolution: The resolution of a digital audio signal refers to the number of bits used to represent each sample. A higher resolution results in higher quality audio, but also requires more storage space and processing power.

- Frame size: The frame size of a digital audio signal is the number of samples that are processed at once by the audio system. A larger frame size can result in more efficient processing, but may also introduce latency.

- Spectrum: The spectrum of an audio signal is a representation of the frequency content of the signal. It can be visualized as a graph showing the amplitudes of the signal's frequencies over a range of frequencies.

- Bin width: In a spectrum, the bin width is the size of the frequency range represented by each data point. A smaller bin width results in a more detailed spectrum, but may also require more processing power.

(b) Figure 1 shows an extract from a Unity C# script that visualises the frequency spectrum of an AudioSource.

```
void CreateVisualisers()
{
    float theta = (Mathf.PI * 2.0f) / (float)AudioAnalyzer.frameSize;
    for (int i = 0; i < AudioAnalyzer.frameSize; i++)
    {
```

```
            Vector3 p = new Vector3(
                Mathf.Sin(theta * i) * radius
                , 0
                , Mathf.Cos(theta * i) * radius
                );
            p = transform.TransformPoint(p);
            Quaternion q = Quaternion.AngleAxis(theta * i * Mathf.Rad2Deg, Vector3.up);
            q = transform.rotation * q;

            GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
            cube.transform.SetPositionAndRotation(p, q);
            cube.transform.parent = this.transform;
            cube.GetComponent<Renderer>().material.color = Color.HSVToRGB(
                i / (float)AudioAnalyzer.frameSize
                , 1
                , 1
                );
            elements.Add(cube);
        }
    }

    // Update is called once per frame
    void Update () {
        for (int i = 0; i < elements.Count; i++) {
            elements[i].transform.localScale = new Vector3(1, 1 + AudioAnalyzer.spectrum[i] *
scale, 1);
        }
     }
```

(i) The generative visual will have a circular shape. The position of each segment in the visual is calculated using the sine and cosine functions applied to the angle theta, which is determined by the frame size of the audio. The x and y positions of each segment are calculated using the sine and cosine of the angle, and the z position is set to 0. The position of each segment is then transformed using the TransformPoint function, which applies the transform of the parent object to the position.

(ii) The orientation of each segment is calculated using the AngleAxis function, which rotates a Quaternion by a specified angle around a specified axis. The angle of rotation is determined by the angle theta multiplied by the index of the segment, and the axis of rotation is the up vector. The resulting Quaternion is then transformed using the parent object's rotation.

(iii) The colour of each segment is determined by the HSVToRGB function, which converts a color in the HSV (hue, saturation, value) color space to a color in the RGB color space. The hue value is calculated as the index of the segment divided by the frame size of the audio, and the saturation and value are both set to 1. The resulting color will be a gradient of colors ranging from red to yellow to green to blue to purple, with each segment being a different shade depending on its position in the gradient.

(iv) The aspect of the visual that is affected by the audio is the scale of each segment. The scale of each segment is determined by the value of the spectrum at the corresponding index, multiplied by a scale factor. To improve the visual so that it is more responsive to the audio characteristics of music, you could increase the sensitivity of the visual to changes in the spectrum by increasing the scale factor or by using a more sophisticated method for mapping the spectrum values to the scale of the segments. You could also consider using additional audio features, such as the beat or the tempo, to affect other aspects of the visual, such as the rotation or the colour.

Question 4

(a) Compare *jobs* with *threads*

(10 marks)

In Unity, jobs and threads are both used to execute tasks concurrently, but they differ in how they are implemented and how they are used.

Jobs:

- Jobs are used in Unity to execute tasks concurrently on a separate thread using the Job System.
- Jobs are created using the [Job] attribute and the job struct, which allows developers to write concurrent code using a simple and intuitive API.
- Jobs are executed on worker threads, which are managed by the Job System.
- Jobs can share data with the main thread and with other jobs, but they must use special constructs, such as the NativeArray class, to do so safely.
- Jobs are well-suited for tasks that can be parallelized, such as data processing or physics simulation.

Threads:

- Threads are a low-level feature of operating systems that allow multiple tasks to be executed concurrently.
- Threads are created using the System.Threading namespace, which provides classes and methods for creating and managing threads.
- Threads are executed on CPU cores and are managed by the operating system's thread scheduler.
- Threads can share data with other threads, but they must use synchronization mechanisms, such as locks or semaphores, to do so safely.
- Threads are well-suited for tasks that require a lot of computing power or that need to run continuously in the background, such as server applications or real-time simulations.

In general, jobs are more convenient to use than threads in Unity because they provide a higher-level API that is easier to work with and that is better integrated with the rest of the engine. Jobs also have built-in support for multithreading, which makes it easier to write concurrent code that takes advantage of multiple CPU cores. However, threads may be more appropriate in some cases, such as when you need fine-grained control over the execution of a task or when you need to use features that are not available in the Job System.

(b) **Figure 2** shows an extract from a procedural animation system that implements a harmonic motion. In porting this code to the C# job system, a Sway Job struct is created that extends IJobParallelForTransform a new class SwayManager is created to manage and schedule the job.

i .What fields should SwayJob and SwayManager have?

(10 marks)

ii . W

hat methods will SwayJob and SwayManager have in order to process, manage and schedule the job?

(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 2

To port the code above to the C# job system, you can create a SwayJob struct that extends IJobParallelForTransform and a SwayManager class to manage and schedule the job.

(i)
Fields for SwayJob:

- float angle: the maximum angle of sway
- float frequency: the frequency of the sway
- float theta: the current angle of sway
- Vectors axis: the axis of sway

Fields for SwayManager:

- List<Transform> transforms: a list of transforms to apply the sway effect to
- float angle: the maximum angle of sway
- float frequency: the frequency of the sway
- Vectors axis: the axis of sway

(ii)
Methods for SwayJob:

- Execute(int index, TransformAccess transform): the method that will be executed in parallel on each transform in the transforms list. This method will apply the sway effect to the transform by rotating it around the specified axis by an angle determined by the current value of theta.
- UpdateTheta(): a method that updates the value of theta based on the elapsed time and the frequency of the sway.

Methods for SwayManager:

- Schedule(): a method that schedules the SwayJob to be executed on the transforms in the transforms list.
- Update(): a method that updates the value of theta for all the SwayJobs and

reschedules them to be executed.

- AddTransform(Transform t): a method that adds a transform to the transforms list.
- RemoveTransform(Transform t): a method that removes a transform from the transforms list.