The Watcher: A Tech Demonstration

Joshua Bridges | 2203030@abertay.ac.uk

Summary

The Watcher is an application demonstration of a particle effects system and three-dimensional sound. The eye in the center is the focal point, where a constant “twitching” sound can be heard, as the eye moves back and forth. In the background, a small planet orbiting a black hole can be observed. This display emanates a sound that invokes a feeling of deep space on a grand scale. All the objects in this scene perform their rotations using DirectX quaternions. Additionally, three distinct particle systems are working simultaneously on the scene. First, a snow system generates a falling snow-like effect to the left of the eye. Second, a fire system works in an opposite direction of the snow to generate a feel of counteracting forces. Last, a space particle system generates from the center of the black hole, dispersing in all directions. These three systems demonstrate the depth of complexity that the implemented particle system can support. In combination with small changes in the lighting and camera behavior, these new systems fashion a scene that draws you in and creates an ominous environment.

Controls

The user has basic controls of the camera in this application. WASD are used to move the camera. W and S move the camera forward and backward respectfully; while A and D move the camera left to right. Users can also use the mouse to rotate the camera. By holding left click, and dragging the mouse, the camera can be manipulated to observe the scene from any angle.

Technical Insight

There are three new systems that will be explored in this technical explanation: the Sound System, Quaternion rotations, and Particle System.

Sound System

Sound effects are created in this application by applying 3D effects to a looping WAV format file sound byte. Each of these effects are mono-channel, and as such, the sound effect needed to be converted on initialization. The implementation of these was greatly reduced by the DirectX audio library and was assisted by the tutorials for implementing sound in the DirectX wiki. In order to create this 3D effect, the emitters for the sound are set to the object of choice. The eye emits the twitching sound and the black hole emits the “space” sound. The listener is then set as the camera. However, with a moving camera the listener’s position needed to be updated every frame. This allows the user to move closer to or further from the emitters and appreciate a drastic change in the sound of the scene.

Quaternion Rotations

Quaternions provide a much more legible version of rotations. The two planets in the background use quaternions to generate some fairly simple rotations. The black hole spins along the Y-axis and the other planet orbits the black hole around the -Y,-Z axis. This allows the object to rotate around a unique axis from the other object in the scene. Additionally, the negative values for the axis makes it so the rotation occurs in the opposite direction of what would normally occur.

The eye uses quaternions to generate a spherical linear interpolation (SLERP) around the sphere. This object uses random generation to create new rotations every few seconds. Once the SLERP is done the eye uses the original orientation and its new orientation to SLERP back to its original angle, facing straight at the camera’s original location. This implementation facilitates quick, smooth rotations showing the eye looking all around the scene.

Particle System

The particle system was implemented using the online rastertek tutorial [2]. The tutorial was originally in DirectX10 so a lot of the implementation was also updating the framework to work with DirectX11.
Particles are generated as quads, using two triangles to define its surface. The triangles are defined by the bottom left, top left, and bottom right vertices; as well as the bottom right, top left, and top right vertices. This specifies the vertices in a clockwise orientation always facing front. However, since only two triangles are used to render the quads, the back face of the particles is not rendered. The user will be able to observe this by moving the camera behind the particles and see them disappear.

These quads were originally rendered using a unique set of shaders. However, this implementation only needs one set of shaders for both the objects and the particles. This allows for some features used with the objects to cross over and be used for the particles as well. Light was the major factor in this decision. The effects of ambient light and a spot light were applied to the remainder of objects in the scene. In this project, it was important for these effects to also apply to the particle system. As a result, the shader with the light implementation was used to render particles to the scene.

Particles are generated frequently, and with a lot of identical particles, the effect would be monotonous. As a result, the system generates each particle with a degree of randomness, with respect to their positions, velocities, and color. This implementation disregards color in favor of the texture on the particle. However, this aspect could later be used to provide a fading effect as each particle ages. The particles then despawn after a time period, to not bog down the scene with too many rendered particles at once.

An enumerated tag allows the designer to specify which type of system they want at any given time. Currently, the system allows for three types: fire, snow, and space. This flexibility can allow for many different behaviors of particle systems, all based on the one framework.

This particle system introduced a unique problem to the outstanding implementation, dynamic buffers in the render pipeline. The system updates the vertex buffer every frame, with the introduction of new particles and the movement of old ones. This implementation solves the issue, by mapping D3D11_MAP_WRITE_DISCARD to the buffer, locking out GPU access until the respective unmapping is accomplished. While the GPU is locked out, the update function can map the new values of the vertex buffer to the address in memory. Once the update is accomplished, the buffer is unlocked, and the GPU renders the updated particle vertices.

Camera and Light

The scene camera is how the user observes the entire scene. Therefore, it was important to have movement seem natural, or be something the user would be accustomed to. Rotation was the key feature for this change. Camera rotation is now bound to the mouse of the user. The camera is implemented using a 3-dimensional vector to specify the orientation. As a result, any rotation simply had to adjust this orientation by adding to it the user’s input, modulated by the camera speed, and the time step between frames.

Lighting also needed adjustment to compensate for the new influx of particles across the scene. Initially, a spot light was used to highlight a small area with an ambient light, providing a small hue to every object. However, the particles generated all over the scene were not getting enough light and would, as a result, show up as dimly light or black regardless of the texture. This was changed to now have the ambient light apply almost white light across the entire screen. In accordance with this change, the spot light was manipulated in the shader to instead cast a darkness on the locations it saturates. This creates an inverse of what the user would normally expect, and compliments the scene well.

Reflection

The scene turned out remarkably well and I am pleased with how this project concluded. Nevertheless, there are a few shortcomings that highlight what could be expanded upon in future work:

1. The readability of quaternions is an important reason for their implementation. However the shader expects the world matrix in a matrix. This means that all of the rotations need to be converted back into matrix format before being fed into the shader. It would be useful for readability if that was able to be implemented in quaternion format.
2. The camera is not using quaternions to move. This was attempted, but the CreateLookAt method gave me issues with getting the orientation to look appropriately using quaternions. Gimbal lock might not be a large issue currently, but it would be important to work around it nonetheless.
3. The particles generated by the systems are only facing one direction and are rendered as paper-thin quads. There are some fun ways this could be changed for the better in the future.
    a. The particles could be changed to be rendered as six quads to form a small cube. This cube would have omnidirectional faces and would allow the user to have a sense of depth from the particles. Since it would be a cube, no convoluted calculations would be necessary as the current measurements could be used to construct the entire cube.
    b. Another, more fun, way would be to have the particles orient themselves towards the camera at all times. This would add to the ominous tone of the scene and fit in naturally. A system could use the orientation of the camera and calculate the inverse of this direction. This would provide an angle that could be used with SLERP or just simple rotations in order to rotate each particle towards the viewer at all times.
4. Render to Texture was unable to be implemented in this version of the scene. This feature would have been able to create a new perspective to compliment the existing scene. A different particle system on a render texture would provide a unique view of the same environment. However, the render to texture was causing two major issues that forced the implementation to be scrapped. First, any orientation of the render texture covered up important aspects of the scene. Second, the particle system displayed in the render texture is scaled down, and, as a result, is almost invisible. This could be implemented in the future by generating the render texture, and applying it to a large object in the scene. Any object with this texture would then appear like a window into a slightly different version of the scene. This version could then have different

particles or an entirely different layout. The only issue would then be redesigning the scene to fit such a massive additional object. One approach might be making a ground that reflects the scene above it but with different features and lighting. Another viable approach would be to have a mirror behind the camera that the user could turn around and view. Regardless of the precise method, implementing this feature would add another layer of depth and visual complexity to the scene.

In conclusion, there are some remaining issues with quaternions and the orientation of some objects in the scene. However, the sound system and particle system both contribute unique features to this application and work phenomenally well with the current scene layout. The flexibility that both of these features provide lay the groundwork for many future innovations and contributions.

Appendix

1. Walburn, C. (2022) *DirectXTX Wiki*, *GitHub*. Available at: https://github.com/microsoft/DirectXTK/wiki/ (Accessed: January 4, 2023).
2. Rastertek (no date) *Tutorial 39: Particle Systems*. Available at: https://www.rastertek.com/dx10tut39.html (Accessed: January 4, 2023).