SETU
Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

4th Year Computer Games Development Project Report

Alan Bolger (C00232036)

Contents

Project Abstract

Video games contain many components that must run as efficiently as possible. Examples of some of these components are physics engines, AI systems, audio systems and level loading/streaming. As most modern games run at frame rates of 60 fps and more, these components have a short window in which to complete a single update cycle. One way of improving performance is by splitting up the work of these components and executing them across multiple threads simultaneously.

The four components that I chose for this project are ray tracing, A* pathfinding, a particle effect generator and a terrain generator. I chose these components as they are not only related closely to game development, but they are subjects that I wished to explore and learn more about.

During my project research, I learned that although multi-threading can be mostly expected to increase performance, if used incorrectly, it can slow down performance. One way of decreasing performance is by constantly creating threads, supplying them with tasks that complete very quickly, then closing the threads once they've completed their tasks then repeating the process. This is why I decided to utilise a threadpool class for this project. A threadpool, once created, will keep a specified number of threads constantly open. Without the overhead of creating and closing threads, tasks can be run as quickly as possible. If more tasks than available threads are added, the threadpool's queue system will prioritise tasks while maintaining efficiency.

Project Introduction

This project features four components that are used as examples of how multi-threading can improve performance.

The first component, ray tracing, is a rendering technique used in computer graphics to simulate the way light interacts with objects in a virtual environment. It traces the path of individual rays of light as they interact with surfaces, simulating effects like shadows, reflections, and refractions with a high degree of realism. Ray tracing is commonly used in modern video games for its ability to create extremely accurate lighting and shadows.

The next component is A* pathfinding, which is a search algorithm commonly used in game development to find the shortest path between two points on a graph or grid. It explores potential paths by considering both the cost to reach a location and a heuristic estimate of the remaining distance to the goal. A* is widely used in video games for NPC movement.

The third component is a particle effect. This is a graphical effect used in game development that simulates the behaviour of individual particles, such as fire, smoke, rain, or explosions, within a virtual environment. These particles are typically rendered as small, independent objects, each with its own attributes like position, size, velocity and colour.

The final component is terrain generation, which involves creating a 2D height map by applying procedural noise functions. These functions add randomness to simulate natural landforms like mountains and valleys. The result is a realistic and varied landscape map that can be used in various applications such as a 3D open world video game, or a 2D role playing game.

Each of these components can be run in single-threaded mode or multi-threaded mode – the total execution time taken is then displayed underneath the threading options section so that performance can be easily compared.

In this project, I research and learn about how each of these four components work, how they are implemented using C++ and SFML, and how to make the components user interactable with the use of ImGui-SFML. Although the main idea of the project is to show examples of how multi-threading can improve performance, I believe that the development of each component and how they were implemented is just as important.

Literature Review

This review is split into four sections as there are four separate components that required researching.

Thread Pool

A thread pool is a crucial concept in concurrent programming, particularly in the context of C++ development. It serves as a managed pool of worker threads that can efficiently execute tasks concurrently, improving the overall performance and resource utilization of applications. In this literature review, we will explore the key aspects of thread pools in C++.

Introduction to Thread Pools

Thread pools are essential for exploiting the full potential of multi-core processors in modern computing. A thread pool typically consists of a fixed number of threads that are pre-created and held in reserve. These threads are assigned tasks to execute concurrently, eliminating the overhead of thread creation and destruction associated with manual thread management. This makes thread pools a valuable tool for enhancing application responsiveness and scalability.

Concurrency in C++

C++ has evolved to support multi-threading and concurrency through libraries such as the C++11 Standard Library's `std::thread`. However, managing threads manually can be error-prone and lead to performance bottlenecks. Thread pools provide a higher-level abstraction for concurrent programming in C++, making it easier to write efficient and scalable multi-threaded applications.

Benefits of Thread Pools

Thread pools offer several advantages, such as efficient resource management. By limiting the number of concurrent threads, they prevent resource exhaustion and contention issues. Thread pools also allow for fine-tuning thread counts to match available CPU cores. This adaptability ensures that applications can utilize available hardware resources optimally.

Task Parallelism

Thread pools facilitate task parallelism, where work is divided into discrete tasks that can be executed independently. Developers can submit tasks to the pool, and the pool takes care of assigning them to available threads. This approach simplifies the process of parallelizing tasks and can lead to significant performance improvements.

Load Balancing

Effective load balancing is a crucial aspect of thread pools. Some thread pool implementations dynamically distribute tasks to threads based on availability and workload. This load balancing ensures that no thread remains idle while others are busy, further optimizing resource utilization.

Scalability and Performance

Thread pools play a pivotal role in enhancing the scalability and performance of C++ applications. They enable efficient utilization of multi-core processors, making it possible to process more tasks concurrently. This scalability is particularly valuable for applications requiring high-throughput data processing or real-time responsiveness.

Conclusion

Thread pools are a fundamental concurrency tool in C++ development. They offer an elegant solution for managing threads, improving resource utilization, and achieving efficient task parallelism. Thread pool libraries in C++ provide a range of features that simplify the development of multi-threaded applications, making them a vital component of modern software development in C++.

Ray Tracing

Ray tracing is a rendering technique that simulates the behaviour of light as it interacts with objects in a virtual scene. It has been a fundamental method in computer graphics for decades, prized for its ability to produce photorealistic images by accurately modelling the physics of light. This literature review explores the evolution of ray tracing, its key components, and its applications in various fields.

Ray tracing was first introduced as a concept in computer graphics by Arthur Appel in 1968. Appel's work laid the foundation for subsequent advancements in the field. In 1986, Turner Whitted expanded on ray tracing by introducing the concept of recursive ray tracing, which enabled more realistic reflections and shadows.

Key Components of Ray Tracing

- **Rays and Intersections:** At its core, ray tracing involves tracing rays of light as they travel through a scene and interact with objects. These rays are cast from the camera's viewpoint and traced to determine the colour of pixels on the image plane.
- **Shading Models:** To simulate how light interacts with surfaces, ray tracing employs various shading models. These models determine the appearance of surfaces based on factors like reflection, refraction, and material properties.
- **Acceleration Structures:** To improve efficiency, acceleration structures like bounding volume hierarchies (BVH) are used to optimize ray-object intersection tests, making real-time ray tracing feasible.

Applications of Ray Tracing

- **Computer Graphics:** Ray tracing is widely used in computer graphics for rendering photorealistic images and videos in movies, video games, and architectural visualization.
- **Scientific Simulation:** Beyond graphics, ray tracing is applied in scientific simulations for tasks like simulating the behaviour of light in optical systems and studying the propagation of electromagnetic waves.
- **Product Design and Engineering:** In product design and engineering, ray tracing aids in simulating the behaviour of light in physical environments, helping designers optimize lighting, materials, and aesthetics.

Challenges and Future Directions

Despite its capabilities, ray tracing is computationally intensive, making it challenging to achieve real-time performance in complex scenes. Recent advancements in hardware, including dedicated ray tracing hardware in GPUs, have improved real-time rendering capabilities. However, there is ongoing research into further optimization techniques and hybrid rendering approaches that combine ray tracing with rasterization.

Conclusion

Ray tracing has evolved significantly since its inception, becoming a cornerstone of computer graphics and finding applications in diverse fields. Its ability to produce visually stunning and physically accurate images has made it a valuable tool in both artistic and scientific endeavours. As hardware and algorithms continue to advance, ray tracing's potential for real-time and interactive rendering continues to grow, promising a future of even more immersive and realistic digital experiences.

A* Pathfinding

A* (pronounced "A star") is a commonly used pathfinding algorithm in the field of artificial intelligence and computer science. It is designed to find the shortest path from a starting point to a goal point while navigating through obstacles in a graph or grid-based environment. This literature review explores the development, principles, and applications of the A* pathfinding algorithm.

A* was first introduced by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968. The algorithm was developed as an enhancement of Dijkstra's algorithm, incorporating heuristics to optimize pathfinding by prioritizing nodes that are closer to the goal.

Key Components of A* Pathfinding

- **Graph Representation:** A* operates on a graph or grid, where nodes represent locations, and edges represent possible transitions between them. It can be applied to a wide range of domains, from video games to robotics and network routing.
- **Heuristic Function:** A* uses a heuristic function (usually denoted as "h(n)") to estimate the cost from a given node to the goal. The heuristic guides the search by prioritizing nodes that are likely to lead to a shorter path.
- **Cost Function:** A* maintains a cost function (usually denoted as "g(n)") that tracks the cost of reaching each node from the start point. It ensures that the algorithm explores the least costly paths.

- **Open and Closed Sets:** A* maintains two sets of nodes: the open set (nodes to be explored) and the closed set (nodes already explored). It iteratively selects nodes from the open set, expanding them and updating the cost and heuristic values.

Applications of A* Pathfinding

- **Robotics:** A* is used in robotics for tasks such as path planning for autonomous robots, ensuring efficient and collision-free navigation in complex environments.
- **Video Games:** A* is a popular choice for implementing NPC (non-player character) movement in video games, enabling characters to navigate game worlds intelligently.
- **Network Routing:** In networking, A* can be applied to find optimal routes for data packets in communication networks, including the internet.

GIS and Map Routing: Geographic Information Systems (GIS) use A* for finding optimal routes on maps, aiding in GPS navigation and location-based services.

Variants and Improvements

Over the years, numerous variants and improvements to A* have been proposed, including D* (Dynamic A*) for dynamically changing environments and Theta* for smoother paths. These variants address specific challenges and further optimize the algorithm's performance.

Conclusion

A* pathfinding remains a fundamental algorithm in various fields, facilitating efficient and intelligent navigation. Its ability to find optimal paths while considering obstacles and constraints has made it a valuable tool in robotics, gaming, networking, and more. Ongoing research continues to refine and adapt A* for specific applications, ensuring its continued relevance in diverse domains.

Particle Effects

Particle effects are a visual technique in computer graphics used to simulate and render various natural phenomena by representing them as a collection of small, independent entities called particles. This literature review explores the development, principles, and applications of particle effects in computer graphics and simulations.

The concept of particle systems for computer graphics was introduced by William Reeves in his 1983 paper, "Particle Systems: A Technique for Modelling a Class of Fuzzy Objects." Reeves described how particles could be used to create dynamic and realistic visual effects.

Key Components of Particle Effects

- **Particles:** Particles are the fundamental units of a particle system. Each particle represents a discrete element, such as a spark, smoke particle, or raindrop.
- **Emission:** Particle systems emit particles from a source or emitter, often with specific initial properties like position, velocity, and colour.
- **Behaviour and Dynamics:** Particles can have various behaviours and dynamics, such as gravity, wind, collision detection, and response. These rules govern how particles move and interact.
- **Rendering:** Particles are rendered using various techniques, including billboarding (2D sprite-based rendering) or 3D geometry. Texture mapping and shaders are used to control particle appearance.

Applications of Particle Effects

- **Video Games:** Particle effects are extensively used in video games to create visually appealing elements like explosions, fire, smoke, rain, and magical spells. They enhance realism and immersion.
- **Special Effects in Film and Animation:** Particle simulations are employed in the film industry to generate complex visual effects, such as realistic fire, smoke, and fluid simulations.
- **Scientific and Engineering Simulations:** Particle systems are used in scientific simulations to model phenomena like fluid dynamics, weather patterns, and the behaviour of particles at the molecular level.
- **User Interface (UI) Enhancements:** Particle effects are utilized in UI design to provide interactive and engaging user experiences, such as animated buttons, hover effects, and transitions.

Variants and Advances

In recent years, advancements in hardware and software have enabled more sophisticated and realistic particle effects. Techniques like GPU particle simulation and volumetric rendering have pushed the boundaries of what can be achieved with particle systems. Additionally, machine learning and AI are being applied to improve the realism and efficiency of particle simulations.

Conclusion

Particle effects have become an integral part of computer graphics and simulations, enriching visual content in various media, from video games to films. The ability to simulate and render complex natural phenomena with particles has made them an indispensable tool in graphics and simulation industries. Ongoing research continues to refine particle system algorithms and techniques, promising even more realistic and immersive visual effects in the future.

Terrain Generation

Terrain generation is a crucial component of computer graphics and simulations, enabling the creation of realistic and varied landscapes for applications like games, simulations, and geospatial analysis. One popular method for generating terrain height maps is through the use of simplex noise. This literature review explores the development, principles, and applications of terrain generation using simplex noise.

Simplex noise, developed by Ken Perlin in 2001 as an improvement over classic Perlin noise, has since become a fundamental technique in procedural terrain generation. It offers several advantages, including reduced directional artifacts and improved performance. Simplex noise was previously patented, but that patent elapsed in January of 2022.

Key Components of Terrain Generation with Simplex Noise

- **Simplex Noise:** Simplex noise is a gradient noise function that assigns values to points in space. It is used to generate pseudo-random and smooth patterns, making it suitable for simulating natural terrain features like mountains, valleys, and plains.
- **Height Map:** A height map is a 2D grid of values representing the elevation or height of points on a terrain. Simplex noise is applied to this grid to determine the height values, creating a realistic landscape.
- **Parameters and Scaling:** Terrain generation using simplex noise involves setting parameters such as octaves, persistence, and lacunarity to control the appearance of the terrain. Scaling factors determine the range of elevations.

Applications of Terrain Generation with Simplex Noise

- **Video Games:** Terrain generation using simplex noise is extensively used in video games to create vast and diverse game worlds. It allows for the generation of natural landscapes, such as forests, mountains, and deserts, that enhance gameplay and exploration.

- **Simulation and Training:** Simulations and training environments often use simplex noise-based terrain generation to create realistic training scenarios for fields like military training, aviation, and disaster response.
- **Geospatial Analysis:** In geospatial applications, simplex noise-based terrain generation aids in creating realistic virtual representations of real-world terrains, supporting studies in geography, urban planning, and environmental science.

Variants and Advances

Advancements in simplex noise-based terrain generation include the integration of other noise functions, like fractal Brownian motion, to add complexity and detail to the terrain. Furthermore, real-time terrain generation techniques and GPU-based algorithms have expanded the use of simplex noise in interactive applications.

Conclusion

Terrain generation using simplex noise has improved the way realistic landscapes are created in computer graphics and simulations. Its versatility and ability to produce diverse terrains with natural features make it an essential tool in game development, simulations, and geospatial analysis. As technology advances, simplex noise-based terrain generation techniques continue to evolve, enabling even more immersive and visually stunning virtual environments.

Evaluation and Discussion

This part is split into four sections as there are four separate components that required evaluation and discussion.

Thread Pool

The 'ThreadPool' class is designed to create and manage a pool of worker threads that can execute tasks concurrently. The constructor is responsible for initializing the thread pool by specifying the number of worker threads to create. Each worker thread operates independently, checking for and executing tasks from a shared job queue.

The thread pool uses C++11 features, including the `<condition_variable>` and `<future>` libraries, to coordinate and manage threads. It provides a simple interface for adding tasks to the job queue and handles the complexities of thread synchronization.

In the constructor, the `start(threadAmount)` method is called, which creates the specified number of worker threads. These worker threads are initialized with lambda functions that define their behaviour. Each worker thread runs in a loop, continuously checking for tasks to execute. It waits for a signal from a condition variable to either process a job or exit if the thread pool is stopping.

The most important function in the 'ThreadPool' class is the `addJob()` method, which allows external code to add tasks to the thread pool. It takes a callable object (typically a lambda function) representing the task to be executed. This callable object is wrapped in a `std::packaged_task` and added to the job queue. A `std::future` is returned immediately, allowing the caller to obtain the result of the task once it's completed.

When tasks are added to the job queue using `addJob()`, they become available for execution by the worker threads. The worker threads, which are continuously looping, compete to pick up and execute these tasks. This concurrency ensures that tasks are distributed among the available threads, making efficient use of system resources.

The stop method is called when the thread pool needs to be shut down. It sets a stopping flag to signal to the worker threads that they should finish executing the tasks in the queue and exit gracefully. The condition variable is used to notify all threads about the stopping condition, allowing them to complete their work.

Although this implementation of a thread pool is relatively simple, it is extremely versatile and is something I will use in many future projects.

Ray Tracing

On completion of the ray tracing component, I was very pleased with the results. I had set out to build a functional ray tracer with multi-threading capabilities, and although there were some additional features I would have liked to have added, I was satisfied with what I ended up with.

The ray tracer implemented in the project offers a range of key features for rendering scenes with spheres of varying material properties. It provides a graphical user interface (GUI) using ImGui, allowing users to adjust rendering parameters such as image dimensions, tile dimensions for multi-threaded rendering, camera position, sphere properties (including radius, colour, emission, transparency, and reflection), and maximum ray recursion depth. The ray tracer supports both single-threaded and multi-threaded rendering. In multi-threaded mode, the image is divided into sections, and a thread pool is employed for concurrent rendering, significantly improving performance. The core rendering process calculates ray intersections with spheres, handles lighting, reflections, and refractions based on material properties, and then writes the pixels to a pixel array/buffer.

Some features I would have liked to have added are texture support and camera rotation. I would also have liked to have implemented a better system for adding and editing spheres in the scene, as the current system involves some trial and error.

The multi-threading aspect of the ray tracer significantly improved rendering time and provided a good example of how parallel processing can improve certain applications.

A* Pathfinding

The AStar class is designed to simplify A* pathfinding operations on a node-based map. To get started, you create an instance of this class and provide essential information, including the map data in the form of a vector of integers, the map's width, and its height. The class constructor handles the initial setup, preparing the necessary data structures for the pathfinding process.

Internally, the class creates a grid of nodes that correspond to the map's layout. These nodes store crucial details, such as their coordinates, whether they represent obstacles, and pointers to neighbouring nodes. This initialization step is crucial as it establishes the fundamental graph structure required for efficient pathfinding.

The core of the A* algorithm is executed through the `run()` method. You specify the starting and ending points on the map, and the algorithm takes care of systematically exploring nodes and their neighbouring nodes while calculating the shortest path. It employs a heuristic function to estimate the distance from the current node to the goal, guiding the search toward the most promising path.

As the algorithm executes, it evaluates nodes based on their local and global goals. The local goal represents the distance from the starting node, while the global goal incorporates the heuristic estimate. Whenever a shorter path to a node is discovered, it updates the node's parent and goal values accordingly.

Once the algorithm completes its operation, you can utilize the `getPath()` method to retrieve the computed optimal path. This method returns a list of coordinates that define the sequence of steps to follow from the starting point to the destination on the map.

For the pathfinding component, I created a 2D dungeon crawler style map, and placed 180 bots in random positions around the map. Once a destination node has been placed, the pathfinding can be started. The bots will then find the optimum path from their current position to the destination, and their path will be drawn over the map.

When using multi-threading, and to obtain optimum performance, I decided to encapsulate an instance of the A* class in each bot. This is great for avoiding any overhead that may arise from using shared data across multiple threads, however, it increases the memory footprint of each bot quite considerably. In the future, when I return to this A* pathfinding component, I will work on making the A* class smaller in size.

Multi-threading performance was, as expected, much quicker than running the pathfinding on a single thread.

Particle Effects

The particle effects component is relatively straightforward and allows the user to move a particle effect around the screen using the mouse. There is a ‘DefaultParticle’ class that handles the particle physics, and in the ‘ParticleEffect’ class, each batch of particles are updated.

The following process takes place when a particle is generated. Firstly, the initial position of the particle is set to a specified starting position. Then, a random angle in the range of 0.1 to 360 degrees is generated. This angle determines the initial direction of motion for the particle.

Next, a random lifetime for the particle is generated, which is determined by a random value between 2.0 and the user set lifetime value. The lifetime value dictates how long the particle will stay alive for.

A timestep value is then calculated. This value can be used to adjust the green and alpha values of the particle on each update, gradually transitioning it from yellow to red and making it fade out.

Additionally, a random speed for the particle is generated, ranging from 0.5 to the user provided speed parameter. This speed determines how fast the particle moves in its initial direction.

Finally, the initial velocity vector of the particle is calculated based on the generated angle and speed, enabling the particle to move in its specified direction. It also sets the alive flag to true, indicating that the particle is active and in motion.

Using single-threaded processing, all batches are queued and processed sequentially. With multi-threading enabled, all batches are run in parallel, and once all threads have finished, then the particles are rendered. Multi-threading with this component provides a huge performance, and you can visually see the particles move much faster the exact moment you click on the multi-threaded option.

My goal when creating this component was to update as many particles as possible while also maintaining a reasonably good frame rate/update time. Overall, I was pleased with the results, as I was able to update 1.5 million particles in around 16ms when using multi-threading, which is just about 60 fps.

If I could change anything about this component, I would maybe add one or two additional particle effects, such as smoke and fire. In the future, when I return to this, I will move the multi-threading from the thread pool to the GPU compute shader to further increase performance.

Terrain Generation

This is the component I was most interested in learning about, as I love the idea of creating expansive worlds using nothing but maths. To create a 2D height map, I implemented a noise class that uses Ken Perlin's simplex noise algorithm.

The noise class initializes a permutation array named `perm`, which is used to shuffle and randomize the gradient vectors used in the noise calculation. It also provides a method called generate to generate Simplex noise. This method takes two input values, x and y, representing the location in the noise space where you want to sample the noise. The algorithm performs several steps to compute the noise value at this point.

The noise generation process begins with a skewing operation, which transforms the input coordinates to align with a grid of equilateral triangles. This step is essential for simplifying the subsequent calculations. Then, the algorithm identifies the grid cell (simplex) containing the sampled point.

Next, the algorithm calculates gradients at the simplex's vertices. These gradients determine the directions of maximum change in the noise function. The code uses a permutation table to look up these gradients based on the cell's position.

After obtaining the gradients, the algorithm calculates a set of weights for each vertex based on the distance between the sampled point and the vertices of the simplex. These weights are used to interpolate between the gradients, creating a smoothly varying noise value.

The final noise value is obtained by summing the weighted contributions from all the simplex's vertices. The result is a value between -1 and 1, representing the generated noise at the specified location.

To ensure the noise value falls within the range 0 and 1, I implemented a normalization step using the `normaliseToRange()` function, which scales the noise value to fit this desired range.

The height map is generated by iterating through each pixel in the designated section, calculating the corresponding index in the height map, and computing the height value. This height value is determined by combining noise functions sampled at different frequencies and amplitudes, influenced by a provided seed. To emphasize terrain features, the height value is raised to the power of 4.0. The result is then stored in the height map, representing the terrain's elevation at that pixel.

The height map is then used to populate a pixel array, where colours are mapped to heights to create the visualisation of a contoured terrain.

Once again, multi-threading performance was much better than single-threaded, allowing terrain maps to be generated very quickly.

Project Milestones

During the project, I didn’t think of having actual project milestones, although I did consider the project to be broken up into four main sections, with each component being a section.

After adding the thread pool class to the empty project, I focused on each component one by one, and only moved onto the next component once the previous one was finished.

Major Technical Achievements

For me, I was really pleased to have implemented a ray tracer and a terrain generator.

Although the ray tracer is relatively simple when compared to professional standard ray tracing software (e.g. Blender), it was really satisfying to see the finished renders, knowing that my ray tracer created those images.

I enjoyed creating the terrain generator as its fun to see your code turn nothing into a realistic looking map. The simplex noise stuff was difficult to understand at first, but it made more sense eventually.

Project Review

Overall, I think the project turned out ok. Looking back, I should have made more of an effort to utilise different methods of multi-threading, instead of just using a thread pool.

Conclusions

The main take from this project is that multi-threaded processing can be considerably faster than single-threaded, but only if tasks can be efficiently split between threads. Tasks that rely on data from external objects or other threads will add overhead and can sometimes make multi-threaded performance slower than single-threaded.

I also learned a lot about ray tracing and terrain generation, which were goals of mine from the start of the project.

Future Work

In the future, I would like to continue working on the ray tracer component, and further develop its features. Ideally, I would like to improve the user interface, and add support for polygonal models with textures.

The terrain generator component is also something that I will be developing further, as I have decided to make a standalone height map generator, and the terrain generator component in this project is a good starting point.

References

Scratchapixel (no date) **Introduction to raytracing: A simple method for creating 3D images**

Available at: https://scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/how-does-it-work.html (Accessed: 09 September 2023).

Peter Shirley, Trevor David Black, Steve Hollasch (no date) **Ray Tracing in One Weekend**

Available at: https://raytracing.github.io/books/RayTracingInOneWeekend.html#addingasphere/ray-sphereintersection (Accessed: 09 September 2023).

Wikipedia (14 August 2023) **Simplex Noise**

Available at: https://en.wikipedia.org/wiki/Simplex_noise (Accessed: 10 September 2023).

Red Blob Games (no date) **Making maps with noise functions**

Available at: https://www.redblobgames.com/maps/terrain-from-noise/ (Accessed: 10 September 2023).