CS 523: Intro to Computer Graphics

Final Project Report

Bharata: An Age of Empires Style Isometric Game

Name: Mihir Kulkarni NETID: mak575

Introduction:

Bharata is a prototype game inspired by classics like Age of Empires, Total War, and other real-time strategy/grand-strategy games, featuring an isometric 2.5D world. This project was developed to demonstrate key graphics and AI techniques. In an isometric game, the scene is drawn using a parallel projection at an angle, giving a 3D look from a 2D viewpoint. The game's world consists of a grid of diamond-shaped tiles drawn in an isometric perspective, with moveable units on the map. The core features implemented include:

- OpenGL Rendering Pipeline: Rendering the game world and sprites using modern OpenGL
- Vertex Buffers and Shaders: Storing geometry in the GPU memory and using GLSL shaders for fast, customizable rendering
- Texture Atlas Management: Combining many sprite images into a single texture for efficient usage and easy animation
- Isometric Grid Generation: Creating a grid of terrain tiles and rendering them in an angled, isometric view.
- Coordinate Transformation: Converting between screen pixel coordinates and grid coordinates.
- Unit Selection: Allowing click-and-drag to draw a selection rectangle to select multiple units at once.
- A* Pathfinding for basic AI: Using A* search algorithm to find optimal paths around obstacles for units when commanded to move.
- Animated Unit Movement: An animation when a unit is selected, which can be extended to movement.

OpenGL Rendering:

OpenGL allows us to harness the graphics processing unit for efficient drawing of many textured shapes each frame. We utilize the graphics pipeline allowing us to supply geometric data to the GPU and use small programs called shaders to control how this geometry is drawn. We:

- Configure an orthographic projection since the game is essentially 2D. This means that no perspective foreshortening – objects always appear the same size.
- The vertex shader runs for each vertex and applies any transformations. We use it to even draw the correct sprite based on an offset and index. This is because we are using spritesheets to render animations.
- The fragment shader runs for each pixel of those shapes and looks up the color from a texture to draw the correct sprite.

OpenGL's pipeline allows for robust and quick rendering. Modern GPUs excel at drawing large number of textured triangles in parallel.

Vertex Buffers and Shaders

To efficiently render many tiles and sprites, Bharata employs Vertex Buffer Objects and custom shaders. A VBO is a chunk of memory on the GPU where we store vertex data. By storing the grid's vertex data in the GPUs high performance memory, we can draw the whole map with a few calls, instead of sending thousands of vertices each frame from CPU, improving rendering speed.

- We generate the vertices for all terrain tiles once, at initialization. Each tile is composed of two triangles. For each triangle we specify 3 vertices, and for each vertex we include the position (x,y coordinates in the game world) and the texture coordinates (u,v) pointing to the correct spot in the terrain texture. All these vertices are packed into a VBO.
- We then use a Vertex Array Object (VAO) to store the description of this data layout, which the GPU will use when drawing.
- In the rendering loop, we bind the VAO and issue a single draw call to draw all tile triangles at once. The GPU then runs the vertex shader for each vertex and the fragment shader for each pixel of the triangles, texturing them appropriately.

Texture Atlas Management

To manage the game's graphics (terrain and unit sprites), we use a texture atlas (also known as a sprite sheet). A texture atlas is one big image that contains many smaller images (sprites) arranged in a grid or tile-set. Using a single large texture for many sprites improves performance and reduces texture switching and draws. In Bharata, we have atlases for:

- In my demo, we might have a simple atlas with just two tile images (e.g., normal ground and a blocked tile). We assign each tile type an index into this atlas.
- A larger sprite sheet that contains frames of the unit's animation. For example, the soldier unit has all its animation frames (like different poses while walking) laid out in one wide image - in our case, 68 frames in a row.

When drawing a particular tile or unit, we calculate the appropriate UV coordinates that correspond to its sprite within the atlas. For instance, if the tile is of type "grass" which is the first image in the atlas, we use UV range [0.0 – 0.5] (assuming two images total horizontally) for that tile's vertices. If it's "water" (second image), we use [0.5 – 1.0] in U to pick the second half of the atlas. Similarly for unit frames: if the unit is currently on animation frame 10 out of 68, we compute the U coordinate offset so that the fragment shader will pick the 10th segment of the unit texture atlas. The shaders then sample the correct sub-image from the atlas for each drawn object.

Isometric Grid Generation and Rendering

The game world is built on a 2D grid which we render in an isometric view. Each grid cell has a type and is associated with a tile sprite from the terrain texture atlas. We procedurally generate the grid – for example, by default all cells might be grass, we can designate some cells as obstacles by marking them with a different type value.

For rendering, each cell at position. The challenge is to position each tile correctly on screen in the isometric layout. In a standard gird, you might draw each cell as a square at x and y multiplied by size. But for isometric, the grid is rotated 45 degrees and scaled. The formula we use for converting grid coordinates to screen coordinates is:

- **x_screen = (col – row) * (tile_width/2) + Xoffset**
- **y_screen = (col + row) * (tile_height/2) + Yoffset**

Offsets are calculated by trial and error for particular screen resolution.

Coordinate Transformation between Screen and Grid

Interacting with an isometric grid (for example, clicking on a tile) requires converting between screen space (pixel coordinates of the window) and grid space (row, col indices of the map). Because of the rotated layout, this conversion is not immediately obvious – we need to invert the projection formula we used for rendering. In Bharata, we derive the grid coordinates from a given screen coordinate by essentially solving the above equations for row and col. The inverse formulas are:

- **col = (x_screen/ (tile_width/2) + y_screen/ (tile_height/2)) / 2**
- **row = (y_screen/ (tile_height/2) - x_screen/ (tile_width/2)) / 2**

In simpler terms, we combine the x and y pixel values to find which grid cell they correspond to.

Unit Selection with Rectangle Drag

A hallmark of RTS games is the ability to select multiple units by dragging a rectangle (often called a "marquee selection"). In Bharata, we implemented unit selection by click-and-drag as follows:

- When the left mouse button is pressed down, we record the starting screen coordinates. As the player drags the mouse, a semi-transparent selection rectangle is typically drawn on the screen to show the selection area.
- When the left mouse button is released, we have the ending coordinates of the drag. These two points define a rectangle in screen space that encompasses some region of the game world.
- We convert the two screen coordinates to grid coordinates (row, col) using the inverse isometric transform discussed above. Because of the isometric orientation, the selection rectangle in screen space will correspond to a somewhat diamond-shaped area in grid space. For simplicity, we can take the min and max of these coordinates to bound the selection region on the grid.
- We iterate over all units in the game, and for each unit we check its grid position. If the unit's row is between rmin and rmax and its column is between cmin and cmax, then that unit lies within the selection bounds. We then mark that unit as selected.

Using this method, the player can select multiple units at once by encompassing them in the drag rectangle. The selected units could then be highlighted in some way – for example, by drawing a selection indicator around them or changing their sprite – which is what we implemented (an animation that indicates the unit is ready for movement). This feature greatly improves the user interaction, allowing group commands and mimicking the behavior of commercial RTS games.

Pathfinding with A* Algorithm

Once units are selected, the player can command them to move to a target location. Simply moving in a straight line isn't sufficient if there are obstacles (like impassable tiles or other units), so we implemented an AI pathfinding algorithm to navigate the grid. We chose the A* (A-star) algorithm, which is a widely used pathfinding method in games for its efficiency and optimality.

How it works:

A* finds the cheapest path from a start node to a goal node on a grid (or graph) by considering both the distance traveled so far and a heuristic estimate of the distance remaining. In our case, each grid cell is a node, and moving to an adjacent walkable cell typically has a cost of 1. We used Manhattan distance (difference in rows plus difference in columns) as the heuristic to estimate distance to the goal. A* maintains two sets: an open set of nodes to explore, and a closed set of nodes already explored. It picks the next node to explore based on the lowest $f = g + h$ score, where g is the cost from the start to that node (distance traveled so far) and h is the heuristic from that node to the goal (distance remaining).

In our implementation, each unit that is given a move command receives a path (a list of grid positions to visit in sequence). We also implemented support for multiple units moving: if a group of units is selected and the player rightclicks a destination, we find not just one goal tile but a set of goal tiles (one per unit, e.g. the clicked tile plus nearby tiles for the rest) so that units don't all stack onto one spot. We used a simple approach of BFS (breadth-first search) from the clicked destination to find the nearest free spots equal to the number of units, then ran A* for each unit to its assigned spot. This way, units spread out to nearby positions at the target area instead of overlapping.

Animated Unit Movement

This is done using the sprite sheet. Using indexes to select the frame to be rendered from the sheet, by cycling through the desired frames in rapid succession for each rendered frame, we can give the illusion of movement and animation. This is done using indices for each frame and a public member variable for each unit called state. This state is incremented till the desired index to get the correct animation.

Conclusion

In summary, Bharata demonstrates the creation of a simple yet rich isometric game environment using computer graphics techniques and basic AI. We successfully built an isometric grid-based world in OpenGL, complete with textured terrain and animated units reminiscent of Age of Empires. Key graphics takeaways include understanding how to use the GPU for efficient rendering (through vertex buffers and shaders) and how to manage many images with texture atlases for performance. We also learned how to handle coordinate systems – converting between the game's world and the screen so that user interactions translate correctly to the game world. On the AI side, implementing A* pathfinding provided insight into how game characters can intelligently navigate, and how algorithms can be optimized for real-time use.

Bharata achieved its goals: it is a functional prototype where you can select multiple units and command them around an isometric map, and the units will find their way and animate as they move. This project not only solidified my understanding of computer graphics programming (from shaders to coordinate math) but also touched on game AI and user interface design in games. It underscores how even a relatively small project can bring together concepts from different domains – graphics, algorithms, and human-computer interaction – to create an interactive visual experience. The knowledge gained here lays a foundation for building more complex 3D games or simulations in the future, as many principles (like efficient rendering and pathfinding) carry over directly.

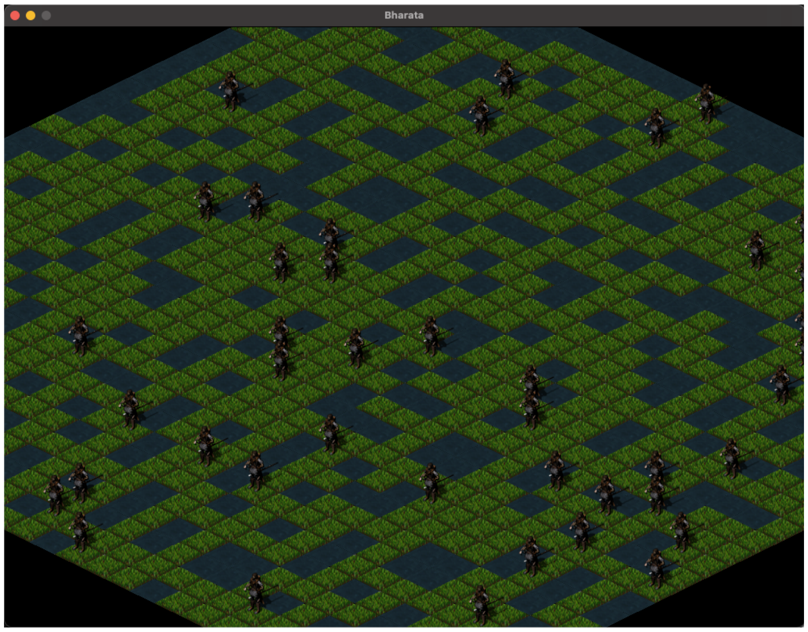
Future Scope

Looking ahead, Bharata can evolve far beyond its current prototype into a richer, more immersive isometric RTS. On the rendering side, adding dynamic lighting—such as a moving sun and point-light sources for torches or buildings—would breathe life into the terrain. Coupling that with normal-mapped tiles or simple bump maps can create convincing depth on otherwise flat textures. At the same time, implementing smooth camera controls (edge-scroll, zoom-in/out, and panning) will let players explore large battlefields, and a minimap overlay will improve strategic situational awareness.

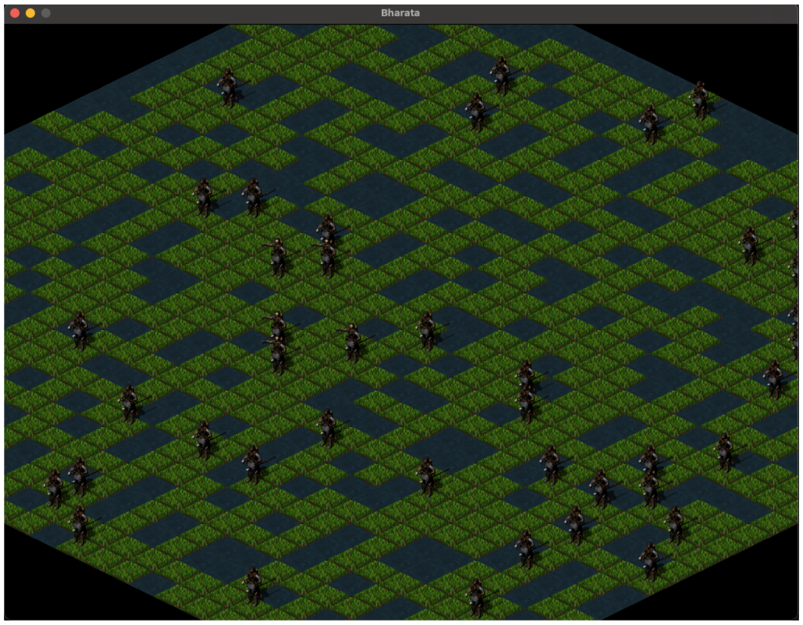
From a gameplay perspective, the grid logic could be extended with true height maps so tiles can rise into hills and cliffs, requiring per-tile depth sorting and occlusion. Unit movement would benefit from formation algorithms that spread selected troops around a target rather than stacking on one cell. On the AI front, hierarchical path-finding (HPA*) will scale smoothly to larger maps, and steering behaviors (e.g. collision avoidance, group cohesion) will give units more natural motion. Adding even a simple economy system—resource nodes, gatherers, and build orders—would bring Bharata closer to a full real-time strategy experience.

Screenshots

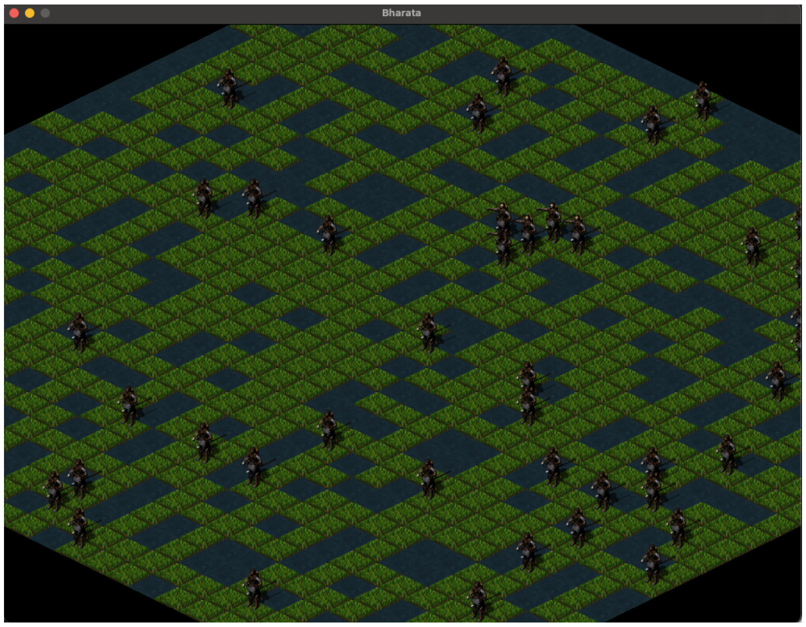
Init State of a random instance:



Some units selected:



Movement to another cell, group movement


Bharata