Documentation for MyCraft

Christopher Gammage

June 19, 2013

1 Components

The system will be composed of the following components.

1.1 Viewport Renderer

The Viewport Renderer handles rendering 3d data to the display. It is responsible for all 3d transformations and optimizations. It handles culling and uses either a 3d rendering library for performance.

1.1.1 Requirements

1. Viewport should be able to render chunks (section 2.2).
2. Viewport should be able to render blocks (section 2.1).
3. Renderer should utilize an abstraction layer to access either OpenGL or DirectX.
4. Renderer should be able to render textures on blocks.
5. Renderer should be able to render non-block items, called Items.
6. Renderer should support particle effects.
7. Renderer should be able to maintain at least 60 fps on a medium end machine. Need to research the spec for minimum machine.
8. Renderer should only make a single render call for each chunk.
9. Renderer should have a lighting system.
10. When building display list for chunk, consider occluded triangles by neighbor edges.
11. Don't render surrounded chunks.
12. Optimize for triangle face merging...

13. Need to decide whether or not to use textures!
14. don't store texture coordinates, generate at runtime.
15. use a texture atlas.
16. use frustum culling.

1.1.2 Rendering

For performance it is important to understand Immediate mode, Vertex Buffers, and Display Lists. During creation of the display list for a chunk, it converts blocks to triangle meshes. Only active blocks are added.

1.2 UI Renderer

The UI Renderer handles the rendering of UI components in 2D on the screen. It is above the Viewport Rendering. It displays widgets, text, and other user HUD related stuff.

1.3 Main Loop

The main loop is the piece of code that ties the rest of the components together. It calls methods of the other components and continuously loops. It is kept in time by the Time Manager.

1.4 UI Manager

The UI Manager handles all user interface interactions except the actual display of user interface. This means the UI manager handles mouse and keyboard events as well as sound output.

1.4.1 Requirements

1. Should modify the camera state.

1.5 Time Manager

The time manager keeps track of both rendering times and game update times. It is responsible for making sure that the game doesn't run too fast and tries to keep it from running too slow.

1.6 Plugin Manager

The plugin manager loads LUA plugins from the plugins folder. It controls initialization, disabling and enabling. It handles dependencies and connects the plugins into the appropriate other components of the system. It should be responsible for making sure LUA code is secure.

1.7 Event Bus

The event bus handles the events of the system. It is a singleton accessible from any component. It uses a publish and subscribe model.

1.8 Logger

The logger is a singleton accessible from anywhere in the system. It does tree based logging. Need to find a file format for the logs that has an easy viewer.

1.9 Scheduler

The scheduler uses the timer to delay release events into the system. It also allows batch events and event chains.

1.10 AI Engine

The AI engine is responsible for updating the game state to reflect artificial intelligence. This means it moves mobs and causes them to act.

1.11 LUA Engine

The LUA engine is a system accessible to plugins which allows them to programatically affect the game state.

1.12 Physics Engine

The physics engine is responsible for all game updates that are not to do with AI. This means gravity, acceleration, water, fire.. etc.

1.12.1 Requirements

1. Responsible for collision detection.

1.12.2 Requirements

1. Is responsible for force loading the chunks surrounding the player in $P_{loadDist}$.

1.13 Network Manager

Handles communication of events across the network.

1.14 Data Manager

The data manager is responsible for the persistence of all world data. It is the master of all game state. It should abstract all data loading to asynchronous processes.

1.14.1 Chunk Manager

Need to decide whether to use array or vector based chunk manager.

Requirements

1. Chunk manager should manage unloading of chunks.
2. Chunk Manager should have a distinction between chunks in visible distance and chunks in render frustum.
3. Chunk Loading should be asynchronous.
4. Chunks should be rebuilt after being modified.
5. Keep track of chunks that are empty.

1.14.2 World Generator

The world generation module is responsible for generating the world. It should create different landscape features based on plugins guidance. Try to use libnoise[2] for terrain generation.

2 Data Types

2.1 Blocks

2.1.1 Requirements

1. Block must have an on-off state
2. Render blocks of size $B_{lwh} = \{l, w, h\}$ in 3D space.
3. Blocks should have $B_{dataSz} = sz$ amount of storage embedded.
4. Blocks should belong in data structure that facilitates fast rendering.
5. Blocks should belong in a data structure that facilitates fast disk save/load.

2.2 Chunks

2.2.1 Requirements

1. Chunks groups of blocks of size C_{lwh}. It is too be decided whether or not a chunk will extend from level 0 to the ceiling of the world.
2. We need to find the best chunk size to use.

2.2.2 Finding the right chunk size

Larger chunk sizes = less rendering overhead, but also larger chunk sizes = slower rebuild times[1].

References

[1] Let's Make a Voxel Engine by AlwaysGeeky. https://sites.google.com/site/letsmakeavoxelengine/home/chunks

[2] Libnoise noise for terrain generation. http://libnoise.sourceforge.net/