

On The Design and Development of 3D Multiplayer Role Playing Games: Lessons Learnt

Matt Honeycutt, Jozef Kaslikowski, Srini Ramaswamy

Software Automation and Intelligence Laboratory

Department of Computer Science and Center for Manufacturing Research

Tennessee Technological University, Cookeville TN 38505

Email: srini@acm.org, Phone: (931)-372-3691

ABSTRACT:

Game programming can be fun and invaluable learning experiences for undergraduate computer science students. It provides significant insights into the various practical issues during program development. The design and implementation of a three dimensional (3D) game is a very challenging task. Students are often inexperienced with game development complexities, have time limitations, and experience difficulty in finding appropriate tutorials and documentation. In this paper we report on several important lessons learnt from this effort that could benefit undergraduate students interested in game programming.

1. Introduction

In this paper, we survey the design and development process for developing 3D, networked computer games (called Hokade Multi-Player Game, or Hokade MPG) for students enthusiastic about game programming. Since many universities lack a complete course in game programming, attempts by interested students in other related courses often results in unrealistic expectations, and subsequent failures. Such failures can be attributed to bad design and lack of forward planning, both critical for successful software development. In many traditional classroom projects, innovative and scalable extensions are often impossible without major rework. Hence students lose the opportunity to learn good practices through reinforcement unless they restart and build upon their failed projects in subsequent classes. Thus students, despite their interests, often go unaware of implementational constraints. The biggest drawback is the lack of introductory texts with step-by-step guidance. Most texts, documentation and tutorials fail to address the issue of how to plan / design a large system; their sample code and demonstrations rigidly focus on individual subsystems, and not on how to integrate and collectively use them in entirety. This paper is a by-product of our attempt to develop a detailed manual for guiding undergraduate students interested in game programming projects within various project-oriented classes. We believe the following features are necessary for realistic multiplayer game programming projects: 3D terrains, client-

server networking, graphical user interface (GUI) system, data file loading/scripting, skeletal character animation, sound and artificial intelligence support.

In our work, we implemented the graphics core first, before working on the 3D terrain system. Then we implemented a basic networking component including necessary features such as thread safety, variable length packet structures, concurrent connection attempts, and automatic reconnection. Then we implemented the game and login servers. Finally, we implemented the GUI system with support for tiling windows and controls, transparency, z-ordering, and included many common GUI controls (edit boxes, text labels, buttons, list boxes, scroll bars, and windows). A scripting system that would allow GUI's to be created and modified with as little effort as possible was also implemented. We considered XML as the most suitable choice. Its hierarchal nature matches GUI design needs, and its plain-text format insures simplicity. To convert an XML GUI description, an XML interpreter was created using recursive-decent parsing. While including support for positional audio, ambient audio, and music could be a good experience if time permits, a simpler solution is to use a singleton "sound manager" that can create and play sound files. Modern hardware improvements have given programmers the ability to write amazing 3D applications that can render in real time. However, there are special considerations that must be made when designing for modern hardware, and it is important to have a firm understanding of the concepts of Hardware T&L, vertex buffers, vertex shaders, and pixel shaders. Without this knowledge, it will be difficult to write efficient code, and impossible to tap the full potential of the hardware.

This paper is organized as follows: Section 2 presents the terrain development issues. Game networking issues are presented in Section 3. GUI related issues are addressed in Section 4. A summary of lessons learnt is presented in Section 5. Section 5 concludes the paper.

2. Terrain Development

In older systems without Graphical Processor Units, or GPUs, two basic methods have been used for terrain rendering - tile-based and static mesh based terrains. In most cases, tile-based systems support only flat terrain or a limited simulation of height. The downside to the static mesh based method is that there is no native Level of Detail (LOD) support. Portions of the mesh that are far from the camera are drawn with the exact same number of polygons as those that are close to the camera, which is unnecessary and wasteful. One of the most popular methods of 3D terrain rendering is ROAM. In ROAM, a mesh representing the terrain is rendered using LOD. The ROAM algorithm subdivides triangles until it reaches the desired level of detail. Prior to fast GPU's, this was a very good solution to terrain rendering.

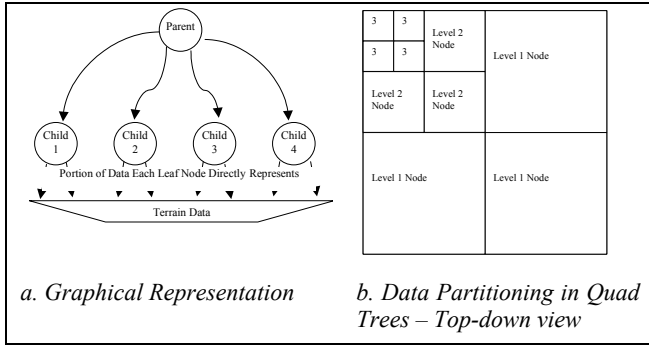


Figure 1. Quad tree data structure

However, this solution is far from optimal on modern 3D hardware. The ROAM algorithm builds a new list of vertices every frame because it performs triangle subdivisions based on distance from the camera. On modern systems, this would require locking a vertex buffer and filling it with new data every frame, which is not very ideal. Ideally, once a vertex buffer has been initially loaded with vertices, it should never be touched again. Also, because ROAM must use the CPU to calculate triangle divisions, the CPU is used heavily while the graphics card is idle. That CPU time could be better spent on other functionality, such as AI, while the GPU handles all aspects of terrain rendering.

2.1 Geometrical Mipmapping

Geometrical mipmapping [1] is a modern algorithm for rendering detailed terrain meshes. The algorithm was proposed as a terrain rendering method that took full advantage of modern graphics hardware. It shares some characteristics with other LOD algorithms (such as ROAM), but it never needs to modify vertex data and therefore can run very quickly on GPU's. Unlike ROAM, Geometrical does not subdivide triangles to reach a desired LOD. Instead, it partitions terrain into equally sized square blocks at load time (with each block stored in its own vertex buffer). At run time, each block is rendered at a level of detail that is appropriate for its distance from the camera. The vertex data for the terrain remains entirely in GPU memory (space permitting), so there is no stall while waiting for the CPU. While the distance calculations still must be performed on the CPU, the calculations are much less complex than those involved with most other methods of terrain rendering.

In the geometrical mipmapping implementation used in this project, a quad tree is used to partition the terrain at load time and to cull non-visible areas at runtime. A quad tree [18, 19] is a specialized tree data structure that is commonly used in graphical applications. In Figure 1a, the lines from the child nodes indicate the bounds of the terrain region they represent. Quad trees are extremely useful for culling non-visible geometry. There is no need to perform per-vertex culling. At each node, only 4 tests are necessary to determine what subset of data must be processed further.

For the purposes of the terrain engine used in this project, only positional data was needed. One may also want to store other

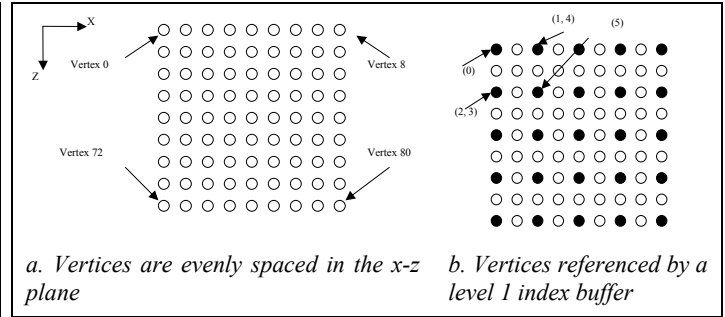


Figure 2. Vertex buffer representation

information about the terrain, such as the type of terrain (grass, snow, dirt, rock), coloring information, and texturing information. Height maps, the map storage method used by Hokade MPG, are a special type of bitmap file that stores height information. Height maps can be generated very quickly from virtually any 2D art package by using Gaussian blurring filters and a mix of black and white spots. The result (if done properly) is realistic looking terrain with a minimal amount of effort. Like all techniques, height maps do have their disadvantages. Designing 3D terrains in a 2D environment can be awkward. There is no way to represent overlapping terrain (as would be caused by ledges, overhangs, and caves). Geometrical mipmapping works best with vertices that are equally spaced (ignoring y) in a grid like fashion. This is exactly the kind of terrain that can be produced with height mapping, so it was a logical choice for this situation.

2.2 Implementation

There are two graphical components needed to render using geometrical mipmapping: vertex buffers and index buffers. One vertex buffer is needed for each terrain block. Each vertex buffer stores all the vertices needed to render its corresponding block at maximum detail. The vertex buffer for each block should be laid out in an identical manner. This implementation of geometrical mipmapping uses indexed triangle lists, where each triangle requires three vertices. The engine can also be implemented using indexed triangle strips, which is discussed later in this paper. In Figure 2a, each circle represents a vertex in a 9x9 terrain block. The corner vertices are marked to indicate their position in the vertex buffer (vertex buffers are essentially one dimensional arrays). In Figure 2b, black vertices represent vertices that were referenced by a level 1 index buffer and are therefore drawn. The values in parenthesis are the index numbers that reference the specified vertex. Four vertices are used to draw two triangles.

There are many steps involved in initialization, but since these steps are only performed once, it will not affect performance once the engine has completed loading. The first step of initialization is

```
void QuadNode.Initialize(MAPDATA **array, int xStart, int yStart, int dimension) {
    if(dimension == BASE_DIMMENSION)
        CreateVertexBuffer();
        LoadDataIntoVertexBuffer();
    else
        this->child1.Initialize(array,xStart,yStart,dimension/2)
        this->child2.Initialize(array xStart+dimension/2,yStart, dimension/2)
        this->child3.Initialize(array,xStart,yStart+dimension/2, dimension/2)
        this->child4.Initialize(array,xStart+dimension/2,yStart+dimension/2,dimesnion/2)
        endif
}
```

Figure 3.Terrain Algorithm Pseudo-code

to load the terrain data for the engine to use. Load the height map into memory, then build a 2D array of vertices (consisting of x, y, and z values for position, and a 3 component normalized normal vector) from this data. Do not use vertex buffers for this step. Note that the height map that is used must be a square bitmap with $(2^n)+1$ pixels per row/column (otherwise it will not be possible to subdivide the map into equally sized blocks). Once the vertices for the terrain have been loaded into memory, the data must be partitioned into equally sized terrain blocks, and then loaded into vertex buffers. For this step, a quad tree is used. Initialization of the quad tree involves a recursive function that takes 4 arguments: a pointer to the 2D array of terrain data, an x and y offset for the upper-left corner of the node's area in the map, and a dimension that indicates the width and height of the node's area of the map. Figure 3 presents the high-level pseudo code of the terrain algorithm used. Ideally, one should choose the width and height of the terrain blocks to be a value that will result in at least 1,000 vertices per block. 33×33 will provide good results, as will 17×17 and 65×65 .

However, care must be exercised when choosing dimensions larger than 33×33 for geometrical mipmapping. Some older hardware cannot allocate buffers large enough to store all the needed vertices for a block of that size. Sizes less than 33×33 will not provide as efficient use of the GPU due to fewer vertices being rendered at once. The final step of initialization for geometrical mipmapping is the creation of the actual mipmaps. For this, index

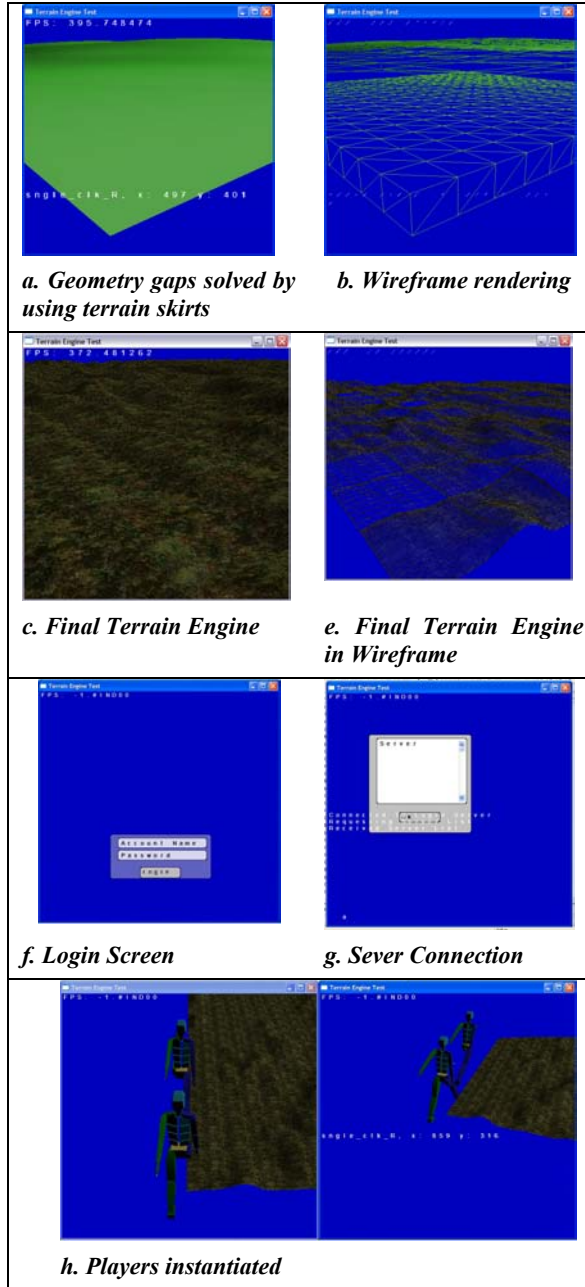


Figure 4. Terrain Rendering

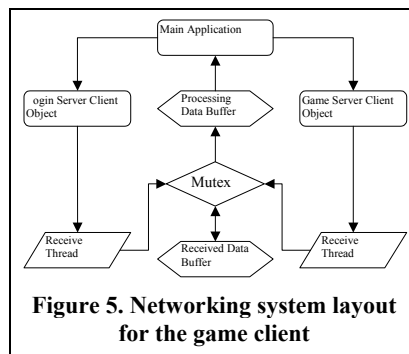


Figure 5. Networking system layout for the game client

buffers are used. One buffer will be created for each level of mipmapping that is needed. For a block with dimensions $(2^n)+1, n+1$ levels of detail can be created. For a 33×33 block $((2^5)+1)$, a total of six levels of mipmapping are needed: 33×33 (full detail), $17 \times 17, 9 \times 9, 5 \times 5, 3 \times 3, 2 \times 2$ (lowest detail). One may choose to only create a subset of the possible detail levels. The size of the index buffers (in number of indices) will depend on the dimension of the block (W) in number of vertices and the level of detail (L). The 0th level of detail is the highest level of detail (meaning that all vertices in the block are rendered), which the Nth level is the lowest level of detail (the terrain block is rendered using only the corner vertices). The following equation can be used to determine how many indices will be used for each index buffer: $NumIndices = 6 * [(W-1)/(2^L)]^2$. Having multiple levels of detail is useless without some mechanism to switch between them. This implementation of geometrical mipmapping bases its choice of detail level on two factors: the height error of the terrain block for each level of detail and the block's distance from the camera. The height error is used to compute a "minimum distance" value that can be compared against the camera's distance to the block. If the distance is greater than the minimum distance value for a lower level of mipmapping, then the lower level is used.

Calculating distance to the camera at runtime is a trivial task, but computing the error in height (and the resulting minimum distance value) is not. Fortunately, one can pre-compute these values at load time (unless allowing deformable terrain, in which case the error will have to be recomputed whenever the terrain changes). Multiple error-factors and minimum distance values must be calculated for each block: one for each level of detail. The pseudo code for this operation is shown in Figure 3. Once the error in height (E) is computed for each level of detail, one can compute the minimum distance value (D^2). The actual equation is discussed in detail in Willem's paper and is fairly complex. A simplified equation is used in Hokade MPG: $D^2 = 0.0625 * E^2 * W^2 * v$. In the above equation, 'W' is the height of the rendering area in pixels, and 'v' is a variable that can be modified to influence the detail level. The higher 'v' is, the further the camera must be from a block before it will change to a lower level of detail. At render time, compute the distance from the camera to each block, square this value, then compare it to the minimum distance value that was pre-calculated at run time.

2.3 Optimizations

While geometrical mipmapping makes excellent use

of the graphics hardware, it does have several weak points. First, it can be quite memory intensive, which makes it difficult to use for large terrains. By using vertex shaders, one can drastically reduce the amount of data that must be stored by storing only a single copy of non-variable data and reusing it for each terrain block. Another way to improve performance (decrease memory usage) is to use a triangle strip instead of a triangle list primitive. With triangle lists, only the first triangle in the strip requires three indices, each additional triangle requires only one additional index (the last two indices from the previous triangle are the first two for the current). Most graphics hardware is optimized for triangle strips.

There is another problem inherit with geometrical mipmapping. When two neighboring blocks are rendered at differing levels of detail, a gap can occur along the shared edge. Though the gap is small (only a few pixels), it is still very noticeable. One possible solution to the problem is to: reorder indices in the higher level block so that it no longer references vertices along the edge that are not being drawn by the neighboring block [1]. While this solution does work, we implement a different solution that was originally suggested by Tom Forsyth (a Microsoft DirectX MVP): terrain skirts. Each terrain block has a small skirt extending downward at the edges (The skirts in Figure 4 were made much larger than needed for illustration). These skirts elegantly hide any gaps without requiring modification of the indices or checking to see whether or not neighbors are being rendered at differing levels of detail. Notice that even at the front-most edges of the terrain, the skirts are invisible except when rendered in wireframe. This is because the skirts use the same normal and the same texture coordinates as the vertices at the edges of the terrain blocks. In Figure 4d, notice the LOD in action as blocks further from the camera and blocks that are flatter are rendered with fewer vertices than those blocks that are bumpy and close to the camera.

In summary, terrain rendering helps to immerse the player in the game world. With modern hardware, truly realistic terrains can be rendered quickly in real-time, but doing so requires a solution that properly uses the hardware to its full potential, such as Geometrical mipmapping. Properly optimized, it can produce excellent results without being resource intensive. We learned several things while implementing the terrain engine. The engine was rewritten (from scratch) on three separate occasions and heavily modified to render using indexed triangle strips instead of triangle lists. Often shortsightedness and poor object interaction were the causes of the majority of the rewrites. Adding better texturing support, using a texturing technique called “splatting” - modified to work extremely well with geometrical mipmapping, could have been a better choice for implementation.

3. Game Networking Issues

While a single player game is fun, online multiplayer games add whole new dimensions to the experience. The multiplayer aspect is accomplished through networking. We will focus on the Internet since our game is designed for larger groups of players.

3.1 UDP Versus TCP/IP

The Internet is built mainly on two types of communication standards, TCP or UDP; each has its own strengths and

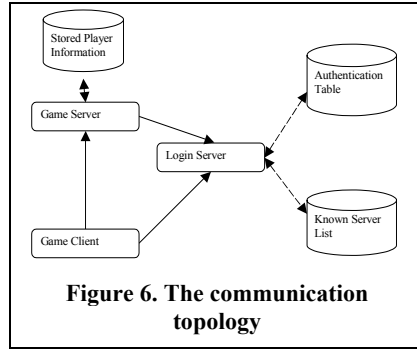


Figure 6. The communication topology

weaknesses. UDP is a connectionless, low delay, low overhead protocol whereas TCP is a connection-oriented larger-delay, larger-overhead protocol. UDP does not rely on the establishment of a virtual connection between both of the communicating computers, the connection is virtual because there is no direct path between the computers and thus the path may be different for each packet of information that flows. UDP is also low delay because there are no guarantees that the information will reach its destination. The low overhead is achieved by the absence of connection

information. TCP on the other hand incurs an immediate delay while establishing a connection between the computers, it has a larger delay because it makes sure all packets arrive and that they are in order. Thus it has a larger overhead due to increased record keeping requirements. Games, in general, have two contradicting network requirements: speed and guaranteed communication.

We chose to use TCP as our transportation protocol due to the huge complexity of handling guaranteed communications over UDP’s best-effort implementation. To accomplish this would require a simulation of TCP on top of UDP, which without intimate knowledge of networking specifics would almost certainly be slower than just using TCP. We have chosen Microsoft’s DirectPlay as an abstraction from the actual sockets and protocol implementation that is normally used to provide networking connection due to the large saving in development time. DirectPlay also provides many features that a bare-bones socket implementation does not. For example, DirectPlay automatically keeps track of network traffic and statistics and allows the statistics to be queried during run-time so that the program can make intelligent gameplay decisions if any players encounter connection problems. DirectPlay also allows runtime switching of packet sending paradigms to allow developers control the communication on a per-packet basis.

3.2 Network Efficiency

The protocol chosen makes little difference if a program is inefficient in its data handling. The total amount of bandwidth needed for messages affects effective gameplay. By using optimal sized packets we attempt to avoid two major, yet subtle, problems that can arise when designing packet structures. It would seem that small packets would be the optimal solution for sending game updates; however, this is not always the case. Sending small packets very frequently leads to a huge increase in the percentage of total bandwidth consumed by packet overhead, overhead that DirectPlay and TCP introduce beyond the actual packet structures. The second issue that must be considered is packet fragmentation; this occurs when packets are too large and must be split into smaller packets so that the hardware and lower-level communication protocols can effectively process them. By splitting one packet into several, the odds that any given packet will be lost or delayed increases.

3.3 Hokade MPG’s Network Design

In our design we took the best pieces from several games to arrive at an elegant and familiar solution. Our design evolved into a client-server system with a separate authentication component. This allows the easy connection of additional game servers while avoiding the complete transfer of authentication information to the

new servers. By consolidating all the authentication details into one separate component, the game server(s) need not concern themselves with these details, thus allowing more processing power and bandwidth to be dedicated to the actual gameplay. Another bonus is that the user has a centralized location from which to choose game servers while the login (authentication) server keeps track of location and address information. However, the centralized nature of the login server could be a bottleneck. The login procedure begins with the game client attempting to connect with a known login server. Once a successful connection is made, the client's user information is authenticated. The game server list is maintained by the login server which has information that identifies server connections and as clients request the server list, it forwards the list to each client. Figure 6 illustrates this functionality.

3.4 Client/Server Implementation

Raph Koster, lead developer of Ultima Online, once said "Never put *anything* on the client[16]. The client is in the hands of the enemy. Never ever ever forget this." Follow his words the client is treated as an input mechanism to the server. While attempting to use the DirectPlay API, many issues were exposed that had not been encountered in previous programs; for example, the use of wide characters and their associated functions. While the use of two byte characters is normal in Java, it is not in C++. Another problem topic can be multithreaded programming. DirectPlay spawns internal threads for both server and client programs and uses them to call callback functions registered to the API. Threads internal to DirectPlay call these callback functions when a packet or other communication is received. And hence the program has to be made thread safe. This can be accomplished by using a WIN32 mutex with two data buffers to receive and process received messages. Since the main application thread must also have access to the received data in the data buffer, a second data buffer was created to copy the received data. This allows the mutex to be released while the data is being processed. In every cycle of the main game loop, the mutex is checked to see if it is free. If the mutex is free, the received data buffer is copied into the processing data buffer, the received buffer is cleared, and its guarding mutex is released. If the mutex is not currently free, the main loop continues and does not block on the mutex. This is done so that the main thread of execution will not stop and wait on the mutex to become unlocked, which could cause efficiency problems due to game updates and rendering delays. After the main thread execution past the mutex check, the data buffer is processed one item at a time.

To identify and process the messages received by the network system, a standardized format and identification system was devised to allow many different types of messages to be sent. First, a base interface was created, which was inherited by all further customized messages. Two issues presented themselves during the design of the message structures: using variable sized packets, and memory alignment of the structure members. The variable sized packet problem is easily solved using an array of length one. For the memory alignment issue, the compiler options were changed to align the code on single byte boundaries instead of the default four. The main reason this was needed was that space for the structures was being allocated based on the raw size of the structures and with four byte alignment, structure data members

were being placed on four byte boundaries and thus were not correctly copied when copying raw memory based on the unaligned structure size.

4. GUI Issues

The main focus of the GUI design was ease of use for both users and developers. GUI rendering can be an extremely complicated task. In our case, we faced a major constraint: rendering the GUI's quickly using Direct3D 8.1, which meant that the GUI's would have to be drawn using polygons. While using polygons created several design issues, Direct3D still provided many benefits. Its native support for alpha blending (rendering polygons with variable transparency) allowed us to include full transparency support in the GUI system. Its texture interface also allowed us to render GUI's where certain portions were totally invisible while others were not, which simplified the rendering of rounded corners on windows and controls. Direct3D supports a "transformed" vertex type that can be rendered directly in screen coordinates. This vertex type was used for all GUI's. In addition to a position, each vertex also contains texture and color information. The texture information helps create a custom "skin" for the GUI's that could be loaded from a texture file such as a bitmap. The color component is needed for alpha blending.

A simple GUI can be created using only four vertices (defining two triangles) and a texture image for that GUI. The top-left vertex should be created at the desired location. The other three vertices should be placed so that the GUI's width and height will match the texture image of the GUI. Failure to do this can cause pixel stretching or other visual artifacts. Some GUI components (such as buttons) can be rendered using very few polygons. A simple GUI needs only two polygons. Rendering such GUI's individually would be a very poor use of the hardware. If possible, all GUI data should be sent to the graphics card at once, or perhaps in several large batches. To achieve this, the base class (CGUI in Figure 7) contains several static members: a dynamic vertex buffer, a dynamic index buffer, and variables to keep track of the current status of the static buffers. These variables are shared by all classes that are derived

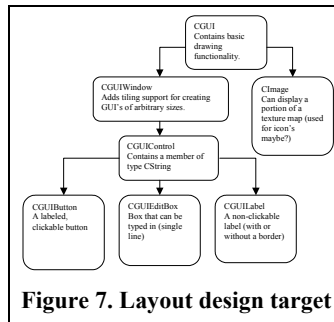


Figure 7. Layout design target

from CGUI, which allows each to store its vertices in a shared location. Once per frame, a "RenderBatch()" method is called that renders all GUI vertices that have not yet been drawn. Because the vertices that define GUI's are likely to change, it is important that a dynamic vertex buffer be used. In some cases, it may be faster to use no vertex buffer at all and render the vertices using one of Direct3D's DrawPrimitiveUP() commands. When using a dynamic vertex buffer, it is important to consider how many vertices will usually need to be rendered per frame. Only part of the buffers should be locked at a time. When those parts are filled, they should be rendered, and the next parts locked.

It is not practical to have a complete image for every type of GUI. Not only would this be memory intensive, but it would also constrain the size of the GUI components. A method is needed to draw GUI's of arbitrary size while minimizing the amount of graphical data that is needed. To accomplish this, we designed a system that can "tile" parts of a GUI component. This allows GUI's of any size and shape to be created. Tiling greatly increases the number of polygons that are needed to render a GUI because each tile must be constructed of two triangles (and therefore four vertices). Because tiling a large GUI can be computationally

expensive, it is best not to perform calculations on every frame. The tiles should be stored so that they can be re-rendered every frame, and new tiles should only be created when the GUI changes location or size.

Finally the windowing system requires event handlers to perform its intended functions. Event handlers are interface functions that are called when an event occurs within a window. These are implemented at a low level within the class hierarchy to ensure that all derived windows conform to the interface. Actual event handlers are implemented as pointers to member functions of each derived class. More details are presented in [20].

5. Conclusions and Lessons Learned

The development of a computer game or a computer game engine can be a very worthwhile task for college students. Students will typically encounter many real world constraints, and thus be compelled to learn new and better programming techniques. However, care must be taken that these goals are not overly ambitious. Such projects are difficult to complete without sufficient motivation. Several issues are not discussed here due to space limitations, these include –fun versus realism, AI and neural network approaches, positional audio etc. The reader is referred to [20] for these issues.

5.1 Design Related Lessons

Game programming projects are complex and all implementation problems cannot be sufficiently planned for. Good OO design is critical to game programming. A throwaway prototyping paradigm (with the expectation that major rewrites may still be required several times during implementation) to overcome design oversights or new ideas is recommended. The most important thing is to focus on a limited number of goals. Implementing a full-featured engine and a game that runs on that engine can be difficult. If the goal were to implement an engine, we would recommend focusing on the design of specific systems that would be needed by a game. If the goal is to implement a game, then use as many existing systems as possible. If possible, use an existing game or graphics engine such as OGRE[21].

5.2 Programming Related Lessons

Several lessons were learned during the implementation of Hokade MPG. Singletons are very useful when designing “manager” type classes[7]. They guarantee that only a single instance of the class can be instantiated, and that this instance is globally accessible. Microsoft includes a powerful DirectX application wizard for Visual Studio with the DirectX SDK [2-6]. It generates a bare-bones application instantly. This application can handle all low-level device and object creation, which frees up development time to focus on other tasks. It also includes several utility classes that implement some useful features (text writing, a frame counter, and DirectInput action mapping are just a few). This application wizard was not used during the development of Hokade MPG, but it could have been a tremendous help. Finally, when implementing a project of this scale in a group environment, it is imperative to use a source code control system of some sort.

6. References

- [1] Boer, Willem H. de. Fast Terrain Rendering Using Geometrical MipMapping. October 2000
<http://www.flipcode.com/tutorials/geomipmaps.pdf>

- [2] Microsoft Corporation. Microsoft DirectX 8.1b SDK C++ Documentation. Oct 2002 <http://msdn.microsoft.com/directx>
- [3] Microsoft Corporation. Microsoft Developer Network. 2002. <http://msdn.microsoft.com>
- [4] Adams, Jim. Programming Role Playing Games with DirectX. Premier Press, 2002.
- [5] McCuskey, Mason. Developing a GUI Using C++ and DirectX – Part I. May 2000. <http://www.gamedev.net/reference/articles/article994.asp>
- [6] McCuskey, Mason. Developing a GUI Using C++ and DirectX – Part II. May 2000. <http://www.gamedev.net/reference/articles/article999.asp>
- [7] Bilas, S. “An Automatic Singleton Utility”, Game Programming Gems. Mass: Charles River Media, Inc., 2000.
- [8] Kh Abdulla, Sarmand. Implementing Skin Meshes with DirectX 8. Sept. 2002 <http://www.gamedev.net/reference/articles/article1835.asp>
- [9] Adams, Jim. Building an X File Frame Hierarchy. Sept. 2002 <http://www.gamedev.net/reference/articles/article1495.asp>
- [10] Adams, Jim. How to parse X files. Sept. 2002 <http://www.gamedev.net/reference/articles/article1493.asp>
- [11] Photon. DirectInput: Converting Scan Codes to ASCII. Sept. 2002 <http://www.gamedev.net/reference/articles/article842.asp>
- [12] Silicon Graphics Inc. OpenGL Website. Dec. 2002 <http://www.opengl.org>
- [13] WWW Consortium. Extensible Markup Language (XML). Dec. 2002 <http://www.w3c.org/XML>
- [14] Misc. Authors. Virtual Terrain Project. Dec. 2002 <http://www.vterrain.org>
- [15] GamesDomain.com. Unreal Tournament 2003 Hands-On. Dec.2002 http://www.gamesdomain.com/gdreview/zones/previews/sep02/unreal_2003.html
- [16] Koster, R. <http://www.legendmud.org/raph/>, Dec. 2002
- [17] Michael, David. Tile/Map-Based Game Techniques: Base Data Structures. Dec. 2002 <http://www.gamedev.net/reference/articles/article837.asp>
- [18] Ferraris, J. Quadtrees. Dec. 2002 <http://www.gamedev.net/reference/programming/features/quadtrees/>
- [19] Johnson, B and Shneiderman, B, “Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures”, *Proc. of IEEE Visualization Conference*, San Diego, CA, Oct., 1991
- [20] M. Honeycutt, J. Kaslikowski, S. Ramaswamy, “A student handbook for developing multiplayer games”, Dept. of CS., TTU, TTUCS-TR-200301S-U001.
- [21] S. Streeting, et. al., Object Oriented Graphics Rendering Engine, <http://ogre.sourceforge.net>