A Collaborative Game Development Tool

Brian Thorpe
University of Colorado at Colorado Springs
bthorpe@uccs.edu

ABSTRACT
This paper describes a tool for collaborative game development. A collaborative game development tool integrates features of traditional game development tools to allow a team to work more effectively. The tools for this project focus on building a game world. These tools become collaborative as they are implemented to work across a shared set of data hosted by a server. Individuals working on the game world can work together by connecting to the game world hosted on the server though a game client. When modifications are made to the game world all changes are persisted to the clients in real time. This allows for teams to work locally or geographically distributed on the same data at the same time.

Author Keywords
Collaborative, Game Development, Revision Control

ACM Classification Keywords
INTRODUCTION
Developing games requires a large number of different participants, including game designers, programmers, writers, musicians, and artists. Whenever there are many participants working on a project, there are overlaps in the work being done. Two major problems result from this overlap: editing conflicts and editing bottlenecks [1]. These problems cause difficulties for the development of game levels. Conflicts and bottlenecks can be alleviated by developing a new system for building game levels. This project researched conventional solutions to parallel distributed development in projects, investigated current solutions for game level development, proposed a new system for game level development, and produced a prototype of the proposed system. The proposed system resolved the problems of parallel development, and provided a low cost solution to smaller game development organizations.

Problem
There is no low cost solution to collaborative game development which provides adequate revision control.

Research
Editing conflicts and editing bottlenecks result from the two most common approaches to revision control. Most revision control systems employ single checkouts with locking, or multiple checkouts with merging, or both. The Revision Control System developed by W. Tichey takes the approach of locking a file [2]. When a file is checked out, the file is subsequently locked preventing other developers from modifying the file. When the developer completes his modifications the file is checked in, allowing other users to check it out and make additional modifications. File locking creates editing bottlenecks. Editing bottlenecks occur when a person wants to work on a piece of code which another person is already working on [1]. Editing bottlenecks can be resolved by allowing multiple people to work on a file at once. An editing conflict is created when the lock is removed and several participants can edit the data. Subsequent check-ins of the file cause the file to first be updated to the base revision. The file is now in conflict with the working version. An approach to resolving this conflict is to merge the two files: the developer can then fix any conflicts in the file that were not resolved automatically.

Editing conflicts are less problematic than having the bottleneck resulting from single checkouts. Generally the solution in software development is state-based merging. Unfortunately, merging cannot be as easily applied to game level data. Game level data is comprised of game elements, art assets, storyline, sound, and many other components. Game level data is commonly stored in a custom binary format. Even if the game level data is stored in a human readable format such as XML, the volume of game level data impedes the ability to merge automatically or manually [1]. This creates conflicts which are difficult to resolve. Thus, allowing game developers to work on the same file in parallel is not a solution.

Two techniques attempt to address this merging problem, Fine Grained Revision Control and Operations Based Merging. A Fine Grained Revision Control System developed by B. Magnusson, U. Asklund, and S. Minor takes the approach of subdividing a program based upon the logical parts of software such as modules, classes, and procedures. Modifications made to a subset of a file are more easily controlled since each logical part of a program is revision controlled [3]. The game level data can be subdivided into smaller pieces allowing a game developer to check out a smaller chunk of data then check the game level data back in later [1]. The subdivision of game level data is similar to the approach taken in the Fine Grained Revision Control System. This reduces the number of bottlenecks, but certainly does not eliminate them in a game level. Subdividing increases the amount of complexity since each subdivided piece must be reconstructed and appear seamless. This could also lead to conflicts between the subdivided pieces. An example in Figure 1 shows how this could happen. Each artist is not aware of the development outside of their subdivision. The triangle is
intersecting another game object. This could be resolved by a single artist working with the top left subdivision. The diamond is centered across two subdivisions. It is ambiguous as to which subdivision the diamond belongs in. The square is in one subdivision but intersects an object in another subdivision. An artist working with each subdivision would not be aware of the conflict. To resolve the conflicts between subdivisions, multiple subdivisions can be checked out. The checkout process creates more bottlenecks, since only one user can work on the data at once. This would allow a single artist to resolve the conflicts in multiple subdivisions, see Figure 2. Operations Based Merging by E. Lippe is another approach to merging instead of state based merging, which looks at merging from a previous state to a new state as discussed above. Operations based merging looks at the set of operations transforming the base version to the new version. The set of transformations on the base set can be used to mitigate some conflicts from strictly state based merging. This approach could reduce some of the conflicts which occur in game level development, but it cannot eliminate them [4].

Figure 1: Potential conflicts in a subdivided game level where each subdivision was developed by a separate artist.

Figure 2: Conflict resolution in a subdivided level. The objects in the subdivisions need to be manually resolved by someone able to see each segment of game level.

As a result neither locking nor merging techniques mitigate the conflicts or bottlenecks associated with the development of game levels. An effective solution to mitigate both bottlenecks and editing conflicts must be developed.

A collaborative environment can be used to mitigate both conflicts and bottlenecks. A collaborative environment allows different users to connect to and work with a shared workspace. In game development, the shared workspace would be the game level. A shared workspace would allow artists to create the game levels, add sound effects, and add art assets, while allowing developers to program scripts for player interaction, and allow game designers and writers to create events, or add storyline. Communication between game designers, artists, and programmers can be facilitated in real time. A server can host the shared space while clients are able to connect and view the current shared workspace on the server. The result is a collaborative game development tool.

Bottlenecks are mitigated within a collaborative game development tool by allowing multiple developers to work on the same level data at the same time. This is achieved by allowing a game designer to create terrain, while another game designer places models in the same part of the game level. Each developer's client sends updates to the game level server and the updates are transmitted from the server to each client. Both modifications happen to the game level at the same time. Any conflict generated from the two clients can be immediately addressed and resolved. This mitigates conflicts by allowing multiple users to see the modifications as they are being made. If a level designer places a tree where another designer thought a building should be, both designers are immediately aware of the conflict and can resolve the problem (see Figure 3). The ability to make modifications in parallel and see the results immediately mitigates the bottleneck and conflict problem.

Figure 3: Resolution of conflicts in figure 1 with a collaborative environment. Multiple artists could resolve conflicts and see results in real time.

In addition to mitigating bottlenecks and conflicts, other benefits emerge. Since the server is hosting the entire game level, there is nothing preventing the collaborative editor
from providing the ability to interact with the game level as a player would while the level is being developed. This allows designers to see the result of the modifications to the game level and then make modifications and changes immediately.

Another benefit is the ability to work remotely and express and share ideas in real time. If a designer has a question about the placement or design of part of the game level, the game level can be modified and suggestions can be made as the designer makes the changes.

The downside to this approach is that there is not the formal check in and checkout process to store the modifications to the game level. There is currently no way to perform the normal operations found in a version control system. This requires a different form of version control [1]. A new form of revision control should be developed to provide revision control features for a collaborative tool.

A collaborative game development environment mitigates the conflicts and bottlenecks associated with traditional check-in and check-out development practices. The collaborative environment also enables game level data to be developed, tested, and discussed amongst a local or distributed team in real time. A revision control system would enhance the utility of a collaborative game development tool enabling the benefits of version control. A collaborative development tool enhances the ability of a team to develop games.

Previous Work
The virtual world Second Life is already a form of collaborative game development for a large continuous virtual world that is editable in real time. Second Life allows people to manipulate the world and see the effects immediately. Objects can be placed or moved. Each player is able to create the game level of their choosing. In turn, all of the other players are able to see other’s creations. This is similar to the collaborative game development tool in that game levels are being created by multiple users simultaneously and all of the users see results immediately.[1][5]

Hero Engine is a collaborative game development tool already on the market providing a collaborative game development solution. Hero Engine allows multiple users to interact and develop a game in real time and see the results immediately. The game levels in Hero Engine are continuous and large. Hero Engine is a platform for large scale game level development targeted at large game development studios. [6]

Proposed Project
This project is the design and implementation of a collaborative game development tool with revision control. The system will provide the necessary tools and features to produce a game level collaboratively with remote users connected to a central server. The server will host the game level, which clients can connect to. Each client will be able to interact with the virtual world as both a designer and a player in real time. Changes made on the server will be created in real time. A revision control system will be developed for the collaborative game development tool.

The game level that this tool creates will be intended for a 3D role playing game where a character explores a virtual environment. The virtual environment the character can explore will contain landscape, buildings, water, caves, and other features. The environment will be described with height maps for the landscape and models for the other physical features. Height maps describe the position of vertices in an evenly distributed two dimensional grid to represent the surface of the landscape. Models can be used to represent anything from buildings to flora and fauna. Directional and point lights will be used to illuminate the environment. Particle systems will be used to represent the point lights for objects such as camp fires and torches. Water will provide reflection and refraction and be represented as a plane. These features will make up the physical game world.

The player will be able to interact with the virtual environment in a variety of ways. The player’s movement will be governed by a rigid body physics system. The player will be able to walk, run, and jump, as well as fall and slide. The player will be able to interact with some of the objects within the virtual environment such as doors. There will also be non-playable characters such as other people and animals that may react to the characters presence. The player will be able to communicate in a fixed manner with characters that may react to the player’s response. Non-playable characters will be able to navigate and interact with the virtual environment and respond to events from the player or other game entities. These features will make up what the player can interact with.

The client and server will be developed in parallel to provide the matching feature sets. The client will be able to interact both as a player and as an editor. This will allow multiple clients connected to the server to both edit and interact with the virtual environment. The client will be able to synchronize with the server upon starting and begin receiving updates from other clients. The server will maintain modifications to entities within the virtual environment. The server will maintain versions of the game world within a revision control system.
PROJECT IMPLEMENTATION
The project implementation covers the general design of the Collaborative Game Development Tool (CGDT), the tools used, and the different components which are required.

Design
The CGDT is composed of four major components: the game client, the game server, the Subversion server, and the web server. Each of these pieces work with one another to create the Collaborative Game Development Tool.

Game World Requirements

World
The world is the entire set of all areas, terrain, models, and any other game data contained within the game. The world contains information about which areas belong to it, the current time of day, and the amount of clouds in the sky. The world contains all of the information in the game and is responsible for managing the areas belonging to the world.

Areas
Areas are subdivisions of the game world. Each area is self-contained and can be loaded without knowledge of any other area in the game. This allows each area to be loaded dynamically at runtime into the world. Areas contain all of the game objects which are inside of the area. Terrain is a special case. Every area contains one terrain game object, but an area can contain any number of other game objects, such as models.

Terrain
The terrain is the game object which corresponds to the surface of the game world. There is one terrain per area. The terrain has the ability to be modified by changing both its height and the textures used to color the ground.

Models
Models consist of any information which has been generated outside of the CGDT. Models are generally objects such as characters, buildings, or trees. Models contain information such as the vertex, texturing, and animation information. These assets are created outside of the CGDT and are then imported. Once imported, models are able to be placed within the game world inside of an area.

Physics
The physics engine provides a physical representation of the data in the virtual environment, allowing for collision detection and rigid body dynamics. The same data used in the rendering engine is sent to the physics engine generating an identical physics environment matching the rendered environment. Using the same data allows the creation of a seamless feel between visual representation and physical representation.

Lighting
Lighting defines the illumination of the game level. This consists of both a global illumination model, and local point light illumination model. Global illumination is light that is applied to the entire game world uniformly. The global illumination typically consists of an ambient light source and a single directional light, which produces shadows. Point lights consist of localized lights that illuminate only a small area surrounding a point. Point lights can produce shadows but require more computation, since the direction of the light is relative to the position of the object casting the shadow.

Water
Water is represented as a singular flat plane which has both reflective and refractive properties. This allows for the
creation of bodies of water simply by manipulating the terrain to be below the water level within the game world.

**Navigation**
Navigation defines areas which game objects can maneuver in and how to perform path finding to and from locations.

**Scripting**
Scripting allows for custom behavior of game objects created through custom coding, which the CGDT can execute. Creating behaviors for game objects allows the game developer to create the interaction and experience for the game player.

**Player Interaction**
The CGDT should allow for the developer to enter the game world and interact from a player perspective, not just a developer perspective. This would allow the designer to test the newly created game level content in real-time while other modifications are being made.

**Game Client Editing Requirements**

**Editing**
Editing is performed on the client through a user interface. Each piece of game data has a customized user interface for manipulating the data.

Terrain editing is done using several tools. There are height modification tools and painting tools. The height modification tools provide four features: raising, lowering, flattening, and smoothing. The painting tools provide three features: paint, erase, and normal. All of the tools have two radii of influence, an inner radius, which paints at the maximum weight, and an outer radius which paints at a linear fall off from the inner radius. There is also a power associated with the paintbrush to determine the maximum influence of the inner radius.

Model manipulation is performed using three tools: a translation tool which allows for manipulation of a model’s position in the game world, a rotation tool that can manipulate a model’s orientation, and a scaling tool which allows for scaling the model to be a different size. The type of tool can be specified by clicking on a model in the world.

The atmosphere can be modified using the GUI interface, the time of day can be static or dynamic, and the length of the day can be increased or decreased. The amount of cloud cover can also be increased or decreased.

Navigation can be created by creating a set of points and nodes. Each point can be connected to form a single navigation node which takes the shape of a triangle. The edges of these triangles can then be used to form a set of points that traverse the mesh. The set of nodes can then be traversed using an A* path finding algorithm.

**Game Client Data Paging**
The game world can contain a large amount of data. The Terrain alone could quickly consume all of the available memory resources on a client machine. Paging allows data to be closed and loaded at runtime. Paging is done per area based on the view of the world on a game client. As the client changes the view, areas are paged in and paged out as necessary to support the current view of the game world.

To maintain synchronization with the server when an area is paged in, it is first checked against subversion for a new version. If the version is current the area is loaded into memory, and then the client begins to synchronize the area with the server.

**Game Client Networking**
Game client includes the communication to the game server and to subversion.

**Game Client to Game Server**
The game client communicates three types of messages to the game server; authentication, editing, and interaction messages.

Authentication messages provide security for the game developers, allowing only certain users to connect and modify the game world. Privileges could be created to distinguish a developer from a tester allowing one to only test the game world, while allowing another access to modify the game world.

Editing messages contain a large number of different edit types. Each type of editing action has information associated with it describing the type of editing modification made. There are unique messages for each component the game level is made of. The messages must be unique since each type of component has a unique set of data that can be modified. Editing messages will also indicate to the server when a piece of the game level must be loaded to accept the edits.

Interaction messages carry information regarding the position of objects and entities when the user is interacting as a player. If multiple users are working together to develop a specific interaction, a single user may be performing the interaction with several other users observing. The information in these messages would carry the position of game objects and their current state. This information could be varied and dependent on the type of scripting created for the game object.

**Game Client to Subversion**
The game client when loaded performs a check against Subversion to determine the current version, just as a software developer would check for updates to the working version before starting development. The game client is able to check for updates to the working game world and updates the necessary areas.

**Game Server**

**Editing**
The editing modifications that occur on the client will be nearly identical to the modifications being applied to the server. The exception is that the server will not contain information necessary to render the game world and will not apply the modifications to data structures which would be rendered.
Game Server Data Paging
Similar to how the amount of data could exceed what the game client can store; the game server can be limited by the amount of available memory as well. The server does not have a rendering of the game world, but must serve the entire game world being manipulated to the clients. The server thus pages data based on the client modifications. If the game client attempts to modify an area that is not loaded, the game server first loads the data then applies the modification. As areas are no longer edited the game server pages the area out. When an area is paged out the data will be lost if it is not saved. Thus, when the area is paged out, the area is also committed to revision control. This allows any client who wants to view the area to quickly check it out from revision control and have the current version.

Game Server Networking
The server manages communication between three separate networked services. The server handles communication between game clients and processes client requests. The server sends requests to the subversion server for checking out and committing game level data. The server verifies the authenticity of game clients with the web level data. The server manages communication between the CGDT and the Subversion server. The server handles establishing and maintaining all of the connections to the clients. When the server sends a request to connect the server, the client is added to a list of connected clients. Secondly, the server asks the client to authenticate itself. Once the client returns the authentication message, the server verifies the authentication against a web server connected to a database. If the client is valid within the database, the webserver returns the client information and the server authenticates the user and adds it to a list of authenticated clients. If the client is not authenticated, it is disconnected and removed from the connected client list. Once the server determines the client is authenticated, the client can begin receiving messages sent by the server or by other game clients passed along by the server. This process continues until the client disconnects from the server by closing the game client.

Tools
Several open source and third party tools have been used in the creation of the Collaborative Game Development Tool.

Microsoft Visual Studio is an Integrated Development Environment (IDE) for developing windows applications in a variety of languages. The language used to develop this project will be C#[7].

Microsoft XNA Game Studio 3.1 provides a managed framework for developing Windows and Xbox 360 games using the DirectX graphics API[8].

Subversion (SVN) is a full-featured version control system that is completely open source. Subversion manages files and the changes made to them over time. This allows Subversion to keep an entire history of modifications being made to the data. Subversion is also capable of running across networks. Subversion is a general system and can be used to provide version control to any type of file[9, 10]. Subversion in the CGDT was used as the backend for handling version control of the game world. A C# wrapper for subversion called Subversion Sharp is used to communicate between the CGDT and Subversion[9].

Lidgren is an open source networking library for C# providing easy packet transmission for UDP between a client and server[11].

Ruby on Rails is an open-source web framework. Ruby on Rails is used within the CGDT to provide a web server which contains information about the users connecting to the CGDT[12].

Sunburn Lighting and Rendering system provides real-time shadows and lighting for the game world. Sunburn uses a deferred rendering model to illuminate the game world [13].

Bepu Physics provides the collision detection, collision response, and rigid body physics within the game world [14].

Tom Shane’s NeoForce GUI Controls provide a library of configurable user interface (UI) elements for XNA. The library contains a variety of elements such as buttons, panels, text boxes, slider controls, and other common user interface controls. The controls mainly used in the development of the CGDT are windows, labels, buttons, sliders, and text boxes. The custom user interface, which is rendered with the game client, provides the ability to overlay the rendering window with controls. Using the normal .Net Winforms library prevents the controls from being rendered on top and must be placed on the outside of the editor. Each approach has benefits and drawbacks, the overlay approach was the solution chosen for the CGDT [15].
RESULTS
The results section describes how the development of the CGDT was accomplished and associates screen shots with each of the different components.

Figure 5: A CGDT Client with a view of a church near a castle reflecting in the water. This is part of the game world created by a client connected to the server.

Game
The game is broken down into a set of components. Each component has a manager that is responsible for managing any game data belonging to it. The components include the terrain, models, physics, navigation, lighting and water.

World
The world within the CGDT is responsible for managing all of the areas in the world and any global properties, such as the atmosphere, which apply to all areas.

The world can be any size. The bounds update based on the smallest bounding box encapsulating all of the areas belonging to the world. One of the major roles the world plays is the loading and unloading of data based upon the current user’s position. The client cannot load all of the game data into memory at once and cannot view the entire game world while being able to maintain acceptable performance for interaction. See the Game Client Paging section and Lessons Learned and Evaluation for more information.

Atmosphere
In addition to managing the areas which are loaded, the world also provides global configuration for lighting and atmosphere. The atmosphere includes the position of the sun and the density of clouds in the sky. Lighting includes the direction light is coming from, the amount of fog in the air, and how the shadows behave.

The atmosphere is composed of three components; the atmospheric scattering model, the clouds, and the night sky. The atmospheric scattering model was implemented based on the implementation by O’Niel [16]. The XNA implementation was developed by Urbano Álvarez [17]. The final model was then integrated with the CGDT and modified to support Sunburn. A few modifications were made to correct errors. The time of day influences the position of the sun, which alters the location of the light entering the atmosphere. The scattering algorithm generates a texture which can then be applied to the surface of a model. The surface the atmospheric scattering texture is applied to within the CGDT is a sky dome. A sky dome is a programatically generated hemi-sphere which is textured underneath to create the appearance of a sky.

Figure 6: An early morning view of the castle within the game world. The atmospheric scattering can be seen around the edges of the mountains. Shadows are being cast upon the castle from the mountains in the foreground.

Night is implemented as a star texture that is stretched across the sky. The moon moves opposite the sun and is displayed as a texture moving across a spherical surface.

Figure 7: A nighttime view of the game world. The moon is high above the castle in the night sky. The moon and stars are two textures applied to the sky dome.

The clouds are imitated as a two dimensional texture that is partially transparent and applied to the sky. A set of noise textures are blended together, which creates the variable appearance of the clouds. The textures are then translated and compared with another noise texture to create the effect of clouds. As the textures move, the clouds change shape resulting in an effect that appears to be dynamic cloud cover. Adjusting the density value of the clouds changes the threshold at which clouds become visible allowing for an increase and decrease of clouds.
The lighting is based upon the configuration of the atmosphere. The orientation of the global directional light is the direction opposite the direction of the Sun. This creates shadows that are lined up with the Sun. The intensity and color of the sunlight are also dependent on the atmospheric scattering model. The amount of clouds can also increase or decrease the amount of light produced by the sun. All of the lighting properties and configuration are done through Sunburn’s interface to the rendering and lighting system.

**Areas**

Areas are subdivisions of the game world. Each area is the same size. Vertical size is not considered in the game world, since the interactions occurring at the surface and height are not nearly as large as the horizontal dimensions. If the game included flying and large sets of models in the sky, a vertical component should be added to account for the amount of information off the ground.

And the center point crosses into another area, it is removed from the current area and added to the new area.

Just as the world is responsible for loading the areas, the area is responsible for loading the terrain and models that belong to it. Areas have several different states: unloaded, loading, loaded, active, active edit. Unloaded indicates the area has only the most basic information attached to it. This includes the areas position, name, and the names of the data files that belong to this area. Once the world determines the area needs to be loaded, the area goes into the loading state. The loading state is where all of the data is read from the hard disk and prepared for further processing. Once the area is finished loading, it enters the loaded state. The loaded state is used to quickly transition the area from being not visible to the user to being visible and rendered. Once the world determines the area needs to be rendered, the active state is enabled. The active state converts all of the loaded data from ram and pushes it to the graphics processor memory for rendering. The last stage is reached only when the player is within a small area, allowing the user to edit up to four areas at once. The areas within this box continually submit new data to the graphics card as the terrain or models are manipulated. The process for how the areas transition between states is covered in the paging section.

**Terrain**

Terrain is the representation of the surface of the game world. The terrain implementation used in this project is a height-map based terrain with geometry mipmapping and texture splatting. A height-map based terrain is simply a grid of vertices that are displaced vertically by a texture containing height information. Texturing is performed by applying multiple textures and blending them with a map of texture intensities. Each terrain area can support up to five unique terrain textures. Terrain rendering can also be very expensive because of the large numbers of triangles being rendered and level of detail implementations that have to be added.

Each area contains a single terrain. The terrain matches the dimensions of the area. In addition to terrain, areas contain any number of models. Models belong to an area based upon the center point of the model. If a model is moved...
Terrain data can become very large very quickly, since each vertex has position, texture coordinates, normal, and morphing information. Each terrain also has indexing information for drawing the triangles at multiple levels of detail. For more information about the memory consumption and performance of terrain, see the Lessons Learned and Evaluation section. For more specific information about the implementation of terrain, please see the Appendix. Explaining the terrain implementation at this point would detract from the overall goal of the paper.

Models
Models are sets of vertices that contain a variety of information about the geometry of an object. There are two built in formats built into the Microsoft XNA Framework FBX and X. The FBX format and the X format can each contain complex geometry, including texture coordinates, color information, normal, skin weights, and animation information [8]. Sunburn also supports these formats as part of its rendering and lighting system. This allows for all of the lighting information to be applied to the models and for correct illumination of the game world.

Water
The water component was created based upon the Sunburn documentation on creating reflection and refractive surfaces. The reflection process requires an additional rendering of the game world as reflected by the surface of the water from the original view. The additional rendering of the game world is an expensive operation and should be disabled on computers with older hardware. To perform this reflection first the view vector is intersected against the water plane. The vector is then mirrored and placed under the water. The new view along this underwater vector is rendered by clipping anything below the water plane. The reflected rendering is then drawn on top of the water surface. The ripples are created by animating a set of normal maps across the surface of the water. The normal information then distorts the reflection. The refraction process can use the original rendering of the game and refract any part of the image below the water surface.

Physics
Every object within the CGDT has a matching set of triangles which describe how the object could interact with game world. All of the game objects are loaded into the BEPU physics engine as triangle meshes. This is supported out of the box by the BEPU physics engine and allows for picking to be performed on the set of triangles. There is currently no dynamic physics built into the CGDT, but the triangle meshes could be used for collision detection against a game agent [14].

Lighting
Lighting has been implemented within the CGDT as an ambient light and a directional light source mapped to a dynamic sky. The directional light and ambient light is provided by Sunburn to illuminate the game world. The directional light contains two major properties; the direction and color of the light. Each light in the Sunburn rendering engine can create shadows that are applied to every game object within the Sunburn rendering system. The shadows are created by using a type of shadow mapping.

Figure 11: A close up view of the castle model within the game world.

Figure 12: A view of the sun reflecting on the water within the CGDT.

Figure 13: Sunlight hitting the castle within the CGDT and casting a shadow on the ground.

The lighting currently exists only on the client side to allow the developer to manipulate lighting conditions to view the game level in different lighting conditions. This allows the global illumination to be manipulated independent of other users.
Point lights are provided as part of the Sunburn rendering and lighting engine. The current implementation of the CGDT does not allow for the placement or modification of these types of light sources. These lights can be positioned within the game world as well as have a radius that they illuminate in the game world. The point light can also have different intensities and color properties applied to it [13].

Navigation

Navigation is currently implemented only as a client side feature. The navigation is an implementation of a navigation mesh with A* navigation applied to it.

Figure 14: A view of the navigation mesh describing where a game object can move and navigate.

The navigation mesh is implemented based upon G. Snooks implementation of navigation meshes [18]. Navigation meshes are useful for simplifying the game world’s geometry, creating a surface game characters are able to walk on. The mesh is formed using triangles; each triangle becomes a navigation node. The triangles along each edge of the triangle are the neighbors for a given node. These nodes can provide information to game agents about which areas are valid to move in and how to move from one location to another using a path-finding algorithm. See Figure 14 for the game world with a navigation mesh laid on top of it.

One difference between this implementation and the implementation described by G. Snook is that the navigation mesh in Snook’s implementation is used to perform collision detection and describes the movement of the character. In this implementation, the navigation mesh only describes the direction the game agent should move to reach a goal. The physics system and collision detection describe the actual movement of the character.

The midpoints on each edge are determined by finding the average of the two vertices forming the line. These midpoints on the edges are used to determine the path through the navigation mesh. Using the midpoints guarantees a path through the middle of the navigation nodes that is least likely to be obstructed by static geometry.

To find neighboring nodes to the current node each edge is examined. If a node has two vertices in common the neighbor is along the edge formed by both vertices.

A* Path Finding

The A* path finding algorithm as described by B. Stout searches a space for the least costly path from a starting location to a goal location by looking at neighboring locations. The A* algorithm repeatedly searches the space looking for the most promising unexplored location. When a location is explored and the algorithm finds the location is not the goal, it searches neighboring locations. A* maintains two lists of locations or nodes. One list contains locations that are open or unexplored and the other list contains locations that are closed or explored. The algorithm picks the next location to explore based upon the current actual cost to the node and an estimated cost to the goal. The sum of the actual cost and the estimated cost determine the next node to be chosen for exploration.

To implement A* on a navigation mesh each triangle is a node, which can be either explored or unexplored. The implementation of A* on the navigation mesh is developed based upon G. Snooks implementation [18]. The path is calculated from the goal node to the start node. The actual cost or g(x) is defined as the distance from the midpoint on the edge coming from the previous node to the midpoint on the edge of the next node. The heuristic cost h(x) is defined as the greatest axis distance from the center point of the current node to the goal node. The path, which is generated from the application of A*, is through the midpoints of the edges of the nodes that define the path from start to goal. See Figure 3 for an example of A* applied to a navigation mesh.

Figure 15: Visualization of the A* path finding algorithm applied to a navigation mesh.

Scripting

Scripting was not implemented due to the additional scope of the feature. See the future work section for a description of how scripting could be implemented in the CGDT.

Dynamic Interaction

Since scripting was not implemented there was nothing to control dynamic interaction with clients on the server. As a result, this feature was not implemented in the CGDT.
Game Client

The game client section describes all of the features unique to the game client.

Paging

Paging is a process that loads and unloads pieces of the game world. As an area is needed, the area is paged in and as an area is no longer needed, it is paged out.

Paging on the game client is performed by loading and unloading areas dependent on the current user’s position and view of the game world. As mentioned in the areas section, an area has four states: unloaded, loaded, active, and, active edit. Depending on the user’s current position within the game world, an area will be in one of these states. The important part of paging areas in and out occurs when the user is changing his or her position within the game world. As a user moves throughout the game world new areas must be loaded as areas are becoming further away must be unloaded.

When paging occurs is determined by multiple thresholds around the user. There is an unload region, a load region, an active region, and an active edit region. The unload region is the largest region and unloads areas from the client. The load region is smaller by the width of a single area. The load region loads areas on the client. The reason there is both an unload region and load region is that it creates a buffer between when something is loaded and when something is unloaded. A user could easily move back and forth if the load region both unloaded and loaded the area, the area would be constantly unloading and loading as the user moved back and forth. To prevent this kind behavior, the unload region extends past the load region, allowing a buffer between when something is loaded to when something becomes unloaded. The loading process is the most expensive process and is performed in a separate background worker thread to prevent any interruption to the users editing experience. Without the thread a noticeable lag is caused when an area needs to be loaded, interrupting the user experience for several seconds. There is a small transition state called loading, which indicates the area is still being loaded inside of the background worker thread. Once the thread loads the area it enters the loaded state.

The transition from loaded state to active state works similarly to the transition from unloaded state to loaded state. The activation region is approximately one area width smaller than the loading region. Once the area is inside the activation region the area is activated. This process creates all of the data necessary to render the area. The area only transfers back to the loaded state when it leaves the loaded region. Again this is done to prevent adding and removing the data necessary to render the area if the user moves back and forth. The processing needed to go from loaded to active is much smaller than the loading process and is performed in the main thread. This has no impact on overall user experience and is usually not noticeable.

Modifications to the game world can only be performed at the active edit state. The active edit state is a small transition mainly for terrain modification purposes. As a result of needing to modify the information stored on the graphics processor, the terrain data must be loaded into buffers, thus duplicating the memory footprint of the terrain. These buffers are then directly loaded into the video memory as they are modified, which replaces the current terrain textures stored there. The active editing region is the smallest region encapsulating only several areas at once. Terrain modification can only be performed on a maximum of four areas at a single time. The active editing state behaves the same as the active state. It is removed from the active editing state when it exits the active edit area. As each area transitions away from the active edit state towards the unloaded state, only the information that was loaded into the previous state is unloaded. This allows the area to still be transitioned from one state to another as fast as possible if the user moves back to the area.

Figure 16: Paging on the Game World. Black Dot Player Location. Red Box Editable Areas to Player. Orange Box Active (visible) Areas for Players. Yellow Box Areas about to be loaded. Purple box areas about to be unloaded

World Editing

The game world contains the set of areas belonging to it and any other information that is stored above the area level. The world is concerned with storing and managing the areas and navigation within the CGDT. Areas can be created or removed using area coordinates. When an area is created a terrain is also created with the area. The world also contains the world navigation information described as a navigation mesh. The CGDT allows for an unlimited number of additional areas to be added to the game world.

Area Editing

Areas contain the set of game objects belonging to it. In the current CGDT, the information stored about a model is only the orientation and position of the game object within the area. Additional information about the area could include a name, special lighting parameters, or anything else that only applies to the area. Currently an area is only modified by adding new game objects to the area.
**Terrain Editing**

Terrain editing is accomplished through the development of several tools. First, terrain needs to be able to be picked selected in three dimensions to determine the intersection point of the cursor on the ground. This is done by performing ray-triangle intersection tests against a ray projected by the cursor position, the view projection, and the triangles in the terrain. When a ground target is determined a ground cursor is added to the ground. The ground cursor is composed of two radii; one is the full power edit radius, which manipulates terrain data by an edit power. The second radii are a linear interpolation from full power to zero power. This creates C1 continuity and produces smooth terrain editing features.

Several terrain height editing tools are required to modify the terrain to create realistic looking surfaces. Tools such as raise, lower, smooth, and flatten provide the necessary features for creating a variety of terrain features, such as rolling hills, steep mountains, cliffs and valleys.

The raise tool raises any vertex in the inner radius by the full power. Any vertex in the outer radius is raised by the full power multiplied by the percent distance from the inner radius to the outer radius. The lower tool is the exact inverse of the raise tool. The smooth tool finds the average height of the vertices in the radius and determines the difference between each individual vertex from the average and moves the vertex towards the average. The outer radius provides a linear fall-off. The flatten tool uses the height of the intersection point with the terrain and moves all of the terrain vertices towards that height. This is similar to the smooth operation, except it does not average the height of all of the vertices using only the one height at the center of the radii.

In addition to height editing tools, texture painting tools allow for the manipulation of ground texturing. The basic tools are paint and erase. A special tool to paint the slopes of hills is called Normal Painting. Paint and erase simply increase or decrease the weighting factors in the splat map for a given texture. The normal paint tool looks at the current slope of the surface and applies texture weighting to the splat map by the given surface normal.

**Model Editing**

The CGDT provides the ability to manipulate the position, orientation, and scaling of models. Other modifications to the shape and texturing of a model require an outside modeling tool. To manipulate the model first the tool, translation, rotation, or scaling, is selected. Once the tool is selected, a model can be selected by picking. Picking projects a ray into the game world based upon the current view projection and the cursor position. If this ray intersects any models, the closest model is returned and is now the selected model. Depending on the tool, a number of gizmos are attached to the model. A gizmo is the term used for an object that is added to the model allowing manipulation. The gizmos that are attached to the model can be picked and dragged to perform the desired operation on the model. Each operation on the model uses the change in the picked position to orient the model differently.

The translation tool adds six gizmos to the model, two arrow gizmos for each axis the model can be translated on. The axis can be either oriented with the game world, or oriented with the models local transform. This allows models to be moved in multiple directions. When a user chooses one of the gizmos along an axis, dragging the gizmo in the direction of that axis moves the object forward or backwards along the selected axis. See Figure 18 for the game object editor controls for translating a game object within the CGDT. Translation can be performed on each axis, red, green, and blue. Currently no axis is highlighted.

**Figure 18: Translating a game object.**

The rotation tool adds three gizmos to the model one for each axis. These gizmos take the form of a ring. Again, this can be in both world coordinate space and local coordinate space. When the user chooses one of the gizmos and drags the cursor along the ring the model will rotate around the axis. This action is similar to turning a steering wheel and seeing the center rotate. See Figure 19 for the game object editor controls for rotating a model within the CGDT. Rotation can be performed on each ring, red, green, and blue. The green ring is highlighted yellow because the cursor is selecting the y-axis for rotation.
The scaling tool adds six gizmos to the model; two box gizmos for each axis the model can be scaled against. Scaling can be done uniformly or along each axis. When a user chooses one of the gizmos along an axis dragging the gizmo away from the model increases the scaling and dragging an arrow towards a model reduces the scaling. See Figure 20 for the game object editor controls for scaling along each axis; red, green, and blue. The Z scaling axis is currently selected and is indicated by the box being highlighted at the end of the blue axis.

**Game Client Networking**

Game client includes the communication to the game server and to subversion. Figure 21, shows the game client’s console view of networking, subversion, and paging messages. The first few messages indicate the game is being updated to revision 815. The next set shows the areas transitioning from different paging states to active editing. This is done because of the proximity of the game client to the area.

![Game Client Log Messages](image)

**Game Client to Game Server**

There are a variety of messages sent from the game client to the game server. There are messages that are sent only from the client to the server, sent from the client and server both, and only sent from the server. The messages only sent from the server are covered in the Game Server to Client section.

The first few types of messages are only sent from the game client to the game server. The keep-alive message is sent periodically to the server informing the server the client is still connected. This message keeps the connection to the client active. The authenticate message communicates the encrypted username and password to the game server. The bye message indicates to the server that client is disconnecting.

The second type of message is communicated by both the client and the server. The only message sent by both the client and server that are not edit messages is the exchanged material message. The exchanged material message contains the public key information necessary for Diffie-Hellman key exchange. The edit messages that are communicated by both the client and server originate on a client and are then executed on the server and forwarded to all other connected clients. The types of editing messages include: area add, area delete, terrain edit, terrain paint, entity add, entity delete, and entity update. The area add message creates a new area on the server just as the area delete removes an area. The terrain edit message contains all of the information necessary to indicate the type of height modification and what parameters were used to modify the terrain. The terrain paint message contains all of the information necessary to indicate the type of paint modification and what parameters were used to paint the
terrain. The entity adds a new game object to the area the player is currently in. This message contains the type of model to create within the game world. The entity delete message removes a model from the game world and contains the unique identifier of the model. The entity update message is sent when the client changes the transform of a model either through translation, rotation, or scaling. This update message contains the new transform for the model that was modified and the unique identifier of that model.

To communicate additional information for new features such as point lights, new messages would need to be created on both the client and server.

Game Client to Subversion

The game client communicates with Subversion only by examining the local version and comparing the local version with the subversion server’s version. If the subversion server has a newer version, the newer version is downloaded and loaded on the client. The download process occurs at two distinct phases. The download will occur when the game client is first loaded, ensuring the local version matches the server’s version. The second download occurs when the game client loads a previously unloaded area. If the area being loaded is an older version than what the subversion server currently has, the new version will be first downloaded from subversion then loaded. The second download allows the game client to get the most recent version of an area before applying any active updates to the area to synchronize. The synchronization process is described in detail in Client Server Synchronization.

Game Server

Game Server Editing

Editing on the game server is identical to modifications being performed on the game client, except the modifications are not applied to information which would be used to render. This reduces the computational complexity on the server, allowing the server to handle multiple editing requests at once.

Paging

Paging is a process that loads and unloads pieces of the game world. As an area is needed the area is paged in, and as an area is no longer needed it is paged out.

Paging on the game server is performed by loading and unloading areas dependent on the current connected client’s modifications to the game world. When a client first begins to modify the game world an edit message is sent to the server. If the modified area is in any state except active, editing the area is immediately loaded and transitioned to active editing. The edit that was sent is then applied to the area. As long as the user continues to modify the area, the area will remain in the active area state. If no modifications are being made the area will remain in active edit state for one minute before it is transitioned back to the active state. After remaining in the active state for a minute the server transitions the area back to the loaded state. After another minute the area will finally be unloaded. During the transition from the loaded state to the unloaded state the server will save the data and submit a new version to the subversion repository. The area will remain unloaded until a user modifies it.

Game Server Networking

The server manages communication between three separate networked services. The server handles communication between game clients and processes client requests. The server sends requests to the subversion server for checking out and committing game level data. The server verifies the authenticity of game clients with the web server. Figure 22 shows the game server log messages, which contain both edit messages, keep-alive messages, and paging messages. Terrain edit messages activate areas; in this example the area 0-0 was activated. Thirty seconds later the area deactivates active editing. Thirty seconds after that the area deactivates. Thirty seconds later the area is completely unloaded.

Figure 22: View of the game server log messages.

Game Server to Game Client

The server has a set of unique messages that are sent to the game client. These messages are only generated by the server. The messages that are forwarded from clients by the server are covered in the Game Client to Game Server section. The server sends a message to the game client indicating whether or not the submitted username and password were valid. This information is contained within the authentication success and authentication denied messages. The display notification message is sent by the server to notify the client of some change in state. The notification sent is a string. This is useful for indicating to a client a change in the network state outside of the normal behavior. An example of this might be a server shutdown message. The last type of message that the server sends uniquely is the Subversion commit message. The Subversion commit message indicates to the clients that an area was recently committed to subversion and any cached modifications to that area should be discarded.

If there are any additions to the type of messages the client sends to the server to edit the world, the server must also
implement these messages to be capable of forwarding the message to all of the connected clients.

Game Server to Subversion
The server has a configuration file, which determines what branch or tag the server is currently hosting. The server reads this configuration file when the server is first started. The server then checks out the branch or version specified in the configuration file. Once the server is started, all of the areas within the game world are unloaded. As soon as a game client begins editing an area, the area is loaded into the active editing state. The area remains in this state as long as the client is editing. When the client stops editing, the area is eventually unloaded. Before unloading, the server commits the current version of the area to Subversion. The server does not communicate with Subversion otherwise.

Game Server to Web Server
A webserver using the Ruby on Rails web framework provides the interface from the game server to a MySQL database. This webserver provides a web browser interface to register as a user for the CGDT. Once the user is validated as a user of the CGDT, the user can log in through the game client. This webserver and database could be extended to store custom information about a user’s preferences, position within the game world, or any other information that would need to be accessible from a terminal running the game client.

The game server is capable of communicating with the webserver across HTTP. The game server constructs HTTP requests containing Java Script Object Notation (JSON) messages. Ruby on Rails provides tools for easily parsing the JSON HTTP requests and returning the requested information from the MySQL database. This allows the game server to look up a connecting game client’s username and password and determine whether or not the game server can authenticate the client and allow them to modify the game world.

Client Server Synchronization
The most difficult part of creating a collaborative tool is to ensure all of the clients and the server are synchronized and the data being worked on is identical between the clients and server. This problem was made even more complex by allowing the client to page information in and out. The condition where one client is modifying areas unloaded by another client is extremely likely since the game world could be vast and multiple areas far apart could be being manipulated by multiple users. Thus, information being modified on one client cannot be manipulated on another client since the information may or may not be loaded on that client. To solve this problem a system of caching updates, leveraging the subversion system, and applying updates to active areas was used.

When a game client first connects to the world, the client first synchronizes the local version with the version stored within the Subversion server. This process is accomplished by performing the subversion update command on the entire subversion tree, which the server has loaded currently. Once the client has acquired the new version the server loads the areas near the user’s position. The areas nearest the user’s position will be loaded into active editing. At the same time the server begins transferring any modifications that have been made to any area since the last revision. This process requires the server to maintain a list of modifications since the last commit for each area. The server can assume a client that just connected has the most recent version from the subversion repository and will send all of the modifications the server has stored for each area since the last revision. The client will have to process all of the messages from the server before the area becomes editable to the user. Areas that are outside of the active editing region will receive the modification messages and queue them in a list until the area becomes active for editing. Once it becomes active, the cached list of modifications will be processed. The client is now initially synchronized for the actively edited areas. Areas outside of this area that are in the active state or loaded state will not be modified until the user moves his client’s view to those areas.

This process doesn’t change much once the client has been connected for a while. The server continues to send updates to the client, which is either processed on areas being actively edited or cached in a list until the area becomes active for editing. Any area in the unloaded, loaded, or active state simply caches the message until the area reaches the active editing state.

The difference occurs when the server determines an area is no longer being modified and commits the version to the repository. The server then clears the list of modifications for the area. When this happens, a commit message is communicated to the client. If the area is in the active editing state but is not being edited by the user, the client will not respond to the change, since it has been synchronized with the server already. An area that is in the loaded or active state will be unloaded and the new version will be updated from Subversion using the update command. The cached list of modifications will also be cleared, since the version in Subversion is the most recent version. Once the area is updated from Subversion it is reloaded into either the loaded or active state depending on what the user’s position in the game world. An area that is in the unloaded state will clear the message cache, because once the area transitions from the unloaded state to the loaded state, the area will first be updated from the subversion repository.
LESSONS LEARNED AND PERFORMANCE
During the development of the CGDT many issues were encountered that changed or impacted the design of the CGDT. Some of the problems encountered impacted design decisions, performance, and the amount of features that were implemented in the CGDT. Overall, the CGDT solved many of these issues, but some of them remain as described in the Future Work section.

Lessons Learned

Design
The design of the CGDT underwent many changes over the development of the tool. Nearly all of the features within the CGDT came from examples and tutorials. These tutorials worked exactly as advertised within their respective demo programs. However, when the demo code was abstracted and applied to a larger system, nearly all of the code had to be completely redesigned. The original designs for the terrain system, scripting system, and dynamic interaction were heavily modified or removed completely. The terrain system was modified heavily to integrate in a collaborative environment, and is covered in detail in the Performance and other Lessons Learned sections. The scripting system was removed entirely as a result of the complexities that were introduced within other systems. The dynamic interaction had to be removed as a result of the loss of scripted behavior.

Maintenance
Maintenance became increasingly difficult as the complexity of the CGDT tool increased. As deadlines approach, less time was spent testing and verifying the functionality of different features. In addition to the amount of code increasing, the number of defects increased as well. These defects impacted the ability to add additional features, as more resources dedicated to finding errors within the current existing features.

Problems within rendering systems can be exceptionally difficult to detect, diagnose, and correct. An example of this occurred with transparent textures. Transparent textures are created by modifying render states. The render states that are modified allow certain pixels within an alpha transparency threshold to be clipped. Enabling these render states causes problems within other rendering systems. As a result of using an external code source to provide much of the rendering capability, attempting to diagnose how the render states affect other subsystems is extremely difficult.

Tutorials
There are hundreds of tutorials and demos on computer graphics and game development. Tutorials are oftentimes a wonderful resource for teaching other people how to implement a technique or perform a task. The problem with tutorials is that normally the technique being taught is simplified to allow for better understanding. This may not seem like a problem at first. The problem occurs when attempting to apply the technique taught within the tutorial to a larger system. Many times the tutorial is not designed to be applied to a larger system and can only stand on its own. Many of the features within the CGDT such as water, navigation, and atmosphere were originally tutorials, which had to be redesigned extensively to be usable within a larger game and game development tool. Tutorials are still a great resource for learning. However, keep in mind that the tutorial should be used as an understanding of a technique and the technique may need to be extended to be applicable to a larger system with many components.

Terrain
Terrain rendering within the CGDT caused a plethora of problems. The rendering, paging, and network communication were all influenced by the additional requirements of terrain. When developing a game development tool, strongly consider the impact of the additional requirements of customizable terrain. Other techniques, such as using custom models generated in third party tools, can provide highly detailed terrain without the expense of creating a customizable terrain editor.

Within the rendering system, terrain had to be created and integrated as a custom piece of the Sunburn rendering system. This required not only knowledge of terrain rendering, but also knowledge of the Sunburn framework. The development process many problems resulted from the customization and integration of terrain into the Sunburn rendering system. As mentioned earlier in the terrain section, the rendering of terrain had to be adapted to work within Sunburn. Custom shader programs had to be developed to account for the lack of multiple vertex streams. In addition to being used within Sunburn, the terrain also needed to be completely dynamic, providing real time updates to the data being rendered on the graphics card. The Performance section has more information on terrain networking and memory performance.

Performance

Rendering
Rendering within the CGDT is handled mostly by Sunburn and is determined mainly by the Sunburn implementation. The rendering system provided by Sunburn is highly optimized and works extremely well for rendering a variety of different things. Factors that affect the rendering speed are the number of shader programs, the number of textures, the number of polygons, and the number of draw calls. Discussing rendering performance is outside of the scope of this paper but a few highlights are the deferred rendering system, the terrain rendering, and the water.

A deferred rendering system is different than the traditional forward rendering model. The major reason the deferred rendering model was chosen for CGDT is the ability to have large numbers of light sources and lighting effects. With a forward rendering system the scene must be redrew with a pass for each light. Within a deferred rendering system the scene is drawn into a variety of different buffers that store different data. Depth, color, normal, shadow, and lighting are just some of the buffers used occasionally.
Each light is drawn into the lighting buffer as a set of pixels with color and depth information. After all of the buffers are created, a final draw call samples each of the buffers and combines the information into the final rendered image. This reduces the amount of redraw of vertex data, but increases the amount of pixel data being written to textures. Unfortunately, the benefits of the deferred rendering system are not seen in the current version of the CGDT as the point light editor was never finished for the CGDT. This would be a small extension to the CGDT to place and communicate the location of point lights within the game world.

The terrain rendering system is extensively optimized and uses a variety of rendering techniques. See the Terrain section in the appendix for details about the terrain renderer.

The water is the most expensive rendering component added to the CGDT. To draw the reflection on the surface of the water, the entire view must be drawn again from a perspective below the surface of the water and looking perpendicular to the cameras view of the water. This redraw uses a smaller resolution view of the game world to limit the size and processing requirements of all of the buffers being created. The easiest way to improve rendering performance would be to turn off the reflection of the water’s surface.

Network
Network performance during the development of the CGDT was not an issue, since the game server has only been run locally. The web server and Subversion server, however, have been hosted on a 3rd party server. There has never been a communication problem with the external servers. The updates, login process, and committing has always occurred very quickly. Specific measures of this performance have not been assessed.

The largest problem with network performance occurred in an earlier version of the CGDT. Most components have simpler data structures, which, when modified; only changed either a transform or a few other properties. This resulted in modifications that occurred across bytes of data. The terrain data when modified occurred across up to megabytes of data. Originally terrain updates were sent as a set of modified vertices to the server which was then forwarded to the clients. The amount of bandwidth usage became excessive when the terrain texturing messages were added. Each terrain has a 512x512 terrain splat texture a single paint message could easily paint an area a quarter of this size sending nearly 512KB per edit. At the time edits were also sent at a rate of 60 times a second. This immediately exceeded the bandwidth the internet connection could support. As a result the terrain modification information had to be changed to an entirely different process. The modification that was used sent only the commands type and the parameters for the command. This reduced the amount of network traffic to a few bytes. The tradeoff was that now the server must perform all of the updates on the terrain. This increased the processing requirements of the server significantly. The server was capable of processing the terrain modifications at a rate of 60 times per second from two clients. In later versions of the CGDT the modification rate was reduced to ten times per second allowing the server to process even more modification requests.

Memory
These problems were mainly a result of the large amount of information contained within the terrain component and the physics system.

The paging system implementation was a major result of this performance issue. Originally the CGDT did not support paging. During the development of the game client a test scene was built programatically with 16x16 areas. Each area consumed memory for the separate textures, the editable copy of the textures, and the textures stored in graphics memory. In addition to the editable textures there are also the textures which are not editable and are typically larger more detailed texture for splotting. These textures can be shared across many areas and are not considered in the memory consumption problem. There are four editable textures per terrain, height, morph height, normal, and splat. The height, morph height, and normal textures are 129x129 textures which are 32-bit resulting in 195KB on disk. The forth texture is the splat map which a 513x513 texture which is 32-bit resulting in 1MB. The total memory per area on disk is 1.198MB. Now factor in the data is stored three times in memory for editing each area consumes 3.59MB. In addition to the memory stored for editing there are additional buffers for indexing triangles which store .375MB of indexed terrain data. There is also the physics system which stores the triangle data and the same indexing information for creating the physics surface. The triangle data accounts for .1875MB and another .375 MB of index data. Including this data in the total each area consumes 4.5 MB of data. At 16x16 areas this consumed a gigabyte of data per terrain. A game has a significant amount of additional data besides terrain to deal with and thus the amount of terrain data had to be limited. Also, it would be quite likely a game world could consume much more than 16x16 areas.

The paging system was a very scalable and effective solution to the memory consumption problem on the client side. The amount of data loaded is dependent on what the user is currently viewing. The amount the character can view is restricted to a small area around the character to limit the amount of memory consumption. In addition to allowing large amount of terrain data, more models can be used as well. Models do not consume as much duplicate data as terrain does because models are not editable in game. Models do however typically have several high detailed textures and depending on the model anywhere one to any number of polygons. For modern games lower polygon model artwork is still desirable to create interactive
frame rates. The physics system creates a triangle mesh from the model as well to perform collision detection which must be loaded on both the server and the client. This duplicates the position vertex and index information for each model, and each instance of the model. The server needs to know what can be collided with if it needs to play an authoritative role in determination of game object or client game play.

The CGDT has not been tested with a large number of users but it can be expected that the server could handle an upper limit of ten concurrent users actively editing the game world at once. The reason the number of concurrent clients is connected is determined by the amount of memory the server would need to use to support ten different users modifying four different areas at once. Modifying forty areas at once would require the server to be actively editing all forty areas. This would put the server’s memory consumption at a high level for a 32-bit application. The memory consumption is dependent on the number of areas loaded and the types of models stored in the active areas. This would also put a restriction on the amount of processing the server could handle; especially if the modifications were large terrain edits being applied by all of the users concurrently. Additional testing outside of the scope of this project would be needed to determine the performance and memory bottlenecks within the game server. New features could be modified, such as a server cluster to allow additional users and larger modifications to occur at once. This feature is talked about more in the future work section.

FUTURE WORK

Component Based Entity System
Adding a component based game entity system would greatly improve the flexibility and design of the CGDT. Instead of each game object being a custom class defining a game object’s behavior, a game object could be described as an empty game entity with a set of components that define the behavior. The game entity would have a set of basic descriptors such as position, orientation, and identification. The game entity would then have a list of game components such as model component, animation component, physics component, and scripting component. The model component would describe the information needed to render the game entity. The animation component would describe the animation of the game model attached to the entity. The physics component would describe how the entity interacts with the game world. The scripting component would be a description of the entity’s behaviors and what actions the entity could execute. The component based model allows for additional customization by using building blocks to describe the game world. This feature allows customization at runtime, as opposed to defining the game behavior at compile time and creating very specific features that are inflexible at runtime.

Scripting
Scripting would be a unique component that would extend the component based model. A scripting component would be able to be added to each game object or entity in the component model. A script would be a set of code that would execute on the game object, defining how the game object would behave within the game world. To create a script within the CGDT the scripts would need to be edited and uploaded to the server and then be available to be applied to the game object. The server would then need to be able to execute the script and update the game object on both the server and the game clients. The .Net Framework provides a compiler object for compiling C# code at runtime and loading a code assembly in memory to execute. This would be the ideal way to create a consistent scripting language, which leverages the other pre-compiled components of the CGDT.

Game Server Cluster
At some point the game world could grow large enough, or enough users may be connected that a single game server would be incapable of hosting the world. Additional servers would be required to handle the memory and processing requirements. This could be accomplished by creating a cluster of game servers to host the game world. A single subversion repository would still exist but the game servers would only be responsible for maintaining modifications on a subset of the areas belonging to the world. A proxy server could be created that all of the game clients connect to. As the game clients send modifications to the proxy server, the proxy would send the modifications to the server responsible for the area being modified. The proxy would then send all of the modification messages to the connected clients.

Improved Revision Control
Currently revision control is performed per area. Each area becomes committed as a client stops editing the area. This causes all the modifications for an area to be committed in a single step. An improvement that could be made to the revision control process would be to create more fine grained revision control of a game area. Just as Magnusson et al. took the approach of subdividing a program based upon the logical parts of software such as modules, classes, and procedures, a game area could be subdivided into logical components such as terrain, navigation, and game objects. Modifications made to a subset of the area could be more easily controlled within the game server and reverting future modifications could be made without modifying the entire setup of an area. This would allow the terrain to be reverted without reverting the game object’s placement.

Content Repository
The content repository would contain all of the game data that is not created using the CGDT. This would include textures, models, and other content, which would normally be included with the client and the server before the applications were started. The content repository would
allow the client and server to synchronize the data that is not part of the CGDT. The content repository would allow users to add new content to the repository, which could then be loaded on the server and downloaded by active clients. This could be viewed as an import feature which would import new content into the CGDT at runtime.

**CONCLUSION**
The CGDT project demonstrated the ability to create a collaborative game development tool that includes revision control. The CGDT is not a complete collaborative game development tool but in its current state could be used to generate the layout for an outdoor game level.

The revision control system works well for maintaining versions of the game world based upon when clients stopped modifying an area. The server can be configured to load specific versions or branches of a Subversion repository allowing traditional version control management to be used in the development of a game world. If branches ever needed to be merged back into the main development branch, problems within areas may result since merging as described earlier is difficult to perform with game data. As a result of areas being subdivisions of the world, areas could be switched in and out with fewer conflicts than merging two areas together. Implementing additional fine level detail control on the CGDT would allow different aspects of areas to be exchanged instead of an entire area.

A surprise benefit of using a version control system was that it allowed clients to easily synchronize to a base version of the game world automatically. There was no additional code necessary to perform file transfers to acquire the current working state of the server. This aspect alone made the revision control system a valuable part of the CGDT.

Overall, the CGDT demonstrated the ability to create a collaborative game development tool with revision control. The CGDT implemented a large number of graphical and game play features required in games, and used all low cost or open source tools to implement the features.

**APPENDIX**

**Terrain**

**Height Map Terrain**

Height map terrains have been used for many years in a variety of applications. The basic idea behind using a height map is to define terrain as a set of height values stored in a rectangular grid. Each height from the height map corresponds to exactly one 3D vertex. Each vertex is evenly spaced in the x-z plane and corresponds to one value mapped to the height map. Each x-z vertex receives the y value from the height map, creating a vertex in 3D space and defining the elevation of the terrain at that point. Each vertex is connected to the neighboring vertices, forming a mesh of triangles. Each pair of four neighboring vertices forms a quad, see Figure 24. The mesh of triangles creates a terrain mesh, which is then distorted by the height map, creating a landscape. An example 32-bit red height map is shown in Figure 25.

**Figure 23**: A 9x9 vertex terrain with a single quad highlighted in orange

**Figure 24**: A 129x129 vertex terrain height map

**Basic Terrain Rendering**

To render this mesh the set of triangles must be drawn by the graphics card. Most graphical frameworks support several ways to draw triangles. Triangles are often drawn simply as a list of vertices or as a list of indices to a set of vertices. Triangles are also defined in three different ways, triangle strips, triangle lists, and triangle fans. The only two formats this paper will look at are triangle strips and triangle lists.

Drawing the mesh without indices uses only a list of vertices; any vertices that are used multiple times are duplicated in the list. A triangle strip is the most efficient way for drawing a set of vertices, because it reduces the amount of duplicate vertex data being sent, see Figure 26. Vertices can often be large in size, containing many floating point numbers and other information. Duplicating this data is expensive, but using a list of indices to map to a single set of vertices reduces this cost.

**Indexing is done by giving a number to each vertex in the set of vertices. The list of indices can then be used to define the triangles as strips, lists, or fans. Duplicating indices in the list is not as expensive as duplicating vertices since a 16-bit integer is half the size of a single floating point number. The smallest vertex data is at least composed of three floating point values and usually four. A vertex can often be composed of more data, such as three**
floats for the position, three floats for the normal, and two floats for the texture coordinates, resulting in a vertex that is 32-bytes in size.

Indices can be defined as either 16-bit or 32-bit. The use of 16-bit indices reduces the amount of information that needs to be stored and transferred to the graphics card for rendering. Unfortunately, using 16-bit indices limits the total number of vertices that can be drawn to a 256x256 vertex terrain. The use of 32-bit indices is twice as expensive. Depending on the terrain rendering process both 16-bit and 32-bit indices have advantages. Indexing vertices and using a list of indices to create triangles is similar to using a list of vertices, see Figure 27.

<table>
<thead>
<tr>
<th>Triangles Defined by Index List</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1,3-Triangle 1</td>
</tr>
<tr>
<td>1,4,3-Triangle 2</td>
</tr>
<tr>
<td>1,2,4-Triangle 3</td>
</tr>
<tr>
<td>2,4,5-Triangle 4</td>
</tr>
<tr>
<td>3,4,6-Triangle 5</td>
</tr>
<tr>
<td>4,6,7-Triangle 6</td>
</tr>
<tr>
<td>4,5,7-Triangle 7</td>
</tr>
<tr>
<td>5,7,8-Triangle 8</td>
</tr>
</tbody>
</table>

Figure 26: Triangles defined as a Triangle List using Indices

Using Triangle Lists increases the amount of data used, but is much easier to work with. The increase in data usage for a list is negligible, since the data is limited to 16-bit or 32-bit integers. Once a list of indices defining each triangle in the terrain mesh is created, the terrain can be drawn. This is the basic way to draw the terrain.

Terrain Lighting
This basic terrain setup only draws a set of triangles with no surface information for providing lighting or diffuse information. To improve upon the terrain surface, normals must be added. Normals for a terrain vertex can be determined by finding the four neighboring vertices. The vertices in both directions are then subtracted creating two vectors. The cross product of the two vectors produces the normal at that vertex, which is then normalized, see Figure 28. Normal information is critical in providing correct shading information for terrain when a light is being applied.

![Figure 27: Normal Calculation for a Vertex V](image1)

![Figure 28: Normal Map](image2)

The normal data is used to determine how much light a surface receives when light is coming at the terrain from any direction. A surface that is perpendicular to the direction of the light receives the most illumination. In Figure 8, it becomes apparent the importance of lighting information and the amount of light the surface is receiving when the light is directly overhead.

![Figure 29: Top Terrain without Normal Data, Middle Terrain with Normal Data, Bottom Terrain with Correct Lighting](image3)

The normal data is stored as a 3D vector and can be added to either the vertex information creating a larger vertex, or as a texture. The texture method uses the red, green, and blue components as x, y, and z values, see figure 29 for an example normal map texture.
Terrain Texturing

Terrain texturing is performed by assigning additional texture coordinates to each vertex in the terrain mesh. Texture coordinates range in two dimensional values defined in u-v coordinates from 0.0 to 1.0. A texture assigned to the terrain will be stretched across the terrain. Assuming the size of the texture being applied to the terrain matches the size of the height map, each vertex will have one texel corresponding to each vertex. Unfortunately, this may result in a stretched texture on the terrain if the terrain is viewed up close. A higher texel to pixel ratio creates a detailed looking terrain texture. When the terrain is viewed up close that texel information is stretched across many pixels, so the resolution of the data is very low, see Figure 31.

Figure 30: Texture Stretching and Pixilation even with Linear Texture Sampling

The first step to improving how the terrain appears close up is to increase the resolution of the terrain texture. A terrain texture that is four times larger than the terrain allows there to be 16 texels for each square of terrain. Increasing the terrain texture resolution significantly improves the quality of how the terrain looks. Increasing the terrain texture resolution, however, can only go so far. For a large piece of terrain to keep the same resolution, larger textures need to be used. For example, a 128x128 piece of terrain needs a 512x512 texture to provide 4x4 texels per quad; a 1024x1024 piece of terrain needs a 4096x4096 texture to provide the same number of texels per quad. There are many approaches to addressing this problem, but increasing the texture size larger than 4096x4096 begins to consume too much memory. To increase the texture resolution further other techniques need to be examined.

One approach is to use texture repetition with a seamless texture. A seamless texture is simply a texture that does not have any seams when it tessellated across left, right, up, and down. The left and top edges match the bottom and right edges, respectively, without a noticeable flaw. Texture repetition allows the texture coordinates assigned to the terrain vertices to be expanded beyond the 0.0-1.0 to 0.0-x.x range where x.x is the number of times the texture will repeat. For example a 128x128 terrain could have a texture which is 128x128 pixels large. If the vertices in the terrain range from 0.0-4.0, the texture will be repeated across the terrain four times in each direction, creating sixteen repetitions. This creates the same number of texels per vertex as using a 512x512 texture. The range can be further increased resulting in more texels per vertex with more repetition across the terrain, see Figure 32. The unfortunate aspect of this approach is that every piece of terrain looks the same. The repetition can also be distracting to the user, so another solution needs to be found.

Figure 31: Terrain Texturing with Repetition using Seamless Texture

Using texture repetition and seamless textures detail texturing was created. Detail texturing is a combination of traditional texturing where a single image is stretched across a terrain to provide color information, while a detail texture is repeated across the terrain to create unique looking color and detail combinations. The detail texture is generally a gray scale image lacking color information. The color information is generally multiplied by the detail texture information, creating different colored detail data. This requires two sets of texture coordinates for each vertex. The first texture coordinate spans between 0 and 1.0 and corresponds to the stretched texture. The second texture coordinate spans between 0 and x.x where x.x is the number of times the detail texture is repeated across the terrain. This allows for unique color information to be spread across the terrain, while creating a high resolution texture appearance generated by the detail texture. This effect works well, but only supports variable color information and a fixed repeated detail texture. Texture still needs to be improved to create realistic looking landscapes.

Charles Bloom developed a technique based upon texture repetition and multi-texturing to create a process called Texture Splatting. Multi-Texturing is the use of multiple textures to create a combined final blended set of texture data. This approach has been modified for many different applications in terrain rendering, but all of these techniques use the same principle of blending multiple layers of repeated textures. Texture Splatting uses a set of repeated textures and a single texture that is stretched across the terrain. The set of repeated textures are blended to create the final result. The single stretched texture is the “Splat Map”, which defines how the other textures are blended. The Splat Map, which is stretched across the terrain, is usually of a higher resolution, providing a minimum of 16x16 texels per terrain quad. The Splat Map uses the four color channels alpha, red, green, and blue to define the blend weights per texel for the other repeated textures, see
Each repeated texture is assigned to one of the four color channels and one base texture is defined. This allows for there to be five textures that can be layered. The base layer is applied in full, and each layer corresponding to one of the color channels is multiplied by the value contained in that channel to determine how much of that texture should be applied, see Figure 34. Each layer can have different scaling value applied to the texture coordinates to independently increase the texture repetition for a specific layer. The result allows for a highly detailed, highly customizable, terrain texture [21]. This is the approach taken in the CGDT. Other techniques of this type use height and surface normal information to automatically determine the blend weights for each texture layer. These approaches create very good looking natural procedural results. I chose the Splat Map approach because it allows for custom painting of the terrain texture, and can still be modified to recreate the same effects seen in the procedural height and surface normal techniques, see Figure 35. This approach creates a great looking terrain, but fails to account for the changes in height that occur across the terrain mesh. So far these techniques only look at texturing in the x-z plane and does not account for texture stretching, which occurs when the difference in height values is very large, see Figure 36. To account for texturing in three dimensions another technique must be added.

Figure 32: Splat Map

Figure 33: Diffuse Splat Textures

Figure 34: Texture Splatting

Figure 35: U, V Texturing

Additional texturing features, such as specular highlighting and bump mapping, could be added to increase the realism of the terrain texturing. Specular highlighting would require the terrain textures to use the alpha component to indicate how much specular light is reflected at that point. This would be the easiest feature to add to the terrain, since it would not require any modifications other than outputting the specular color in the terrain shader. Bump mapping would require additional code to modify the depth of the pixel being output, as well as additional textures, describing the depth of the surfaces for each terrain texture. Bump mapping would be significantly more processing and memory intensive.
Level of Detail
To achieve level of detail in terrain rendering, several techniques have been developed. There are continuous techniques that create a continuous level of detail across the terrain. These techniques are Quad Tree based terrain and Real Time Optimally Adapting Mesh (ROAM) terrain. There are also chunked techniques that have level of detail in segments, which are not continuous. This paper focuses on the chunked level of detail technique called Geometry Mip-mapping. The major difference between continuous and chunked techniques is the speed at which the indexing can be accomplished. Chunked techniques are generally faster than continuous techniques, since a large amount of the indexing data can be preprocessed. The reduction in the dynamic calculation of indices is a greater improvement than the improvement from the better approximation in continuous techniques [23].

Geometry Mip-mapping is the application of the concept of mip-mapping to terrain. Mip-maps are used in images to generate lower resolution images, which can be swapped for higher resolution images when the lower resolution image has the detail necessary for approximating the higher resolution image. Mip-maps are generated by removing every other pixel in an image. This same concept and process can be applied to the terrain mesh by removing every other vertex. Geometry Mip-mapping results in the process seen in Figure 38 [19].

The benefit of geometry mip-mapping is that it reduces the number of triangles that must be processed as the terrain is being drawn. The level of detail is determined by the distance from the camera position. The further a piece of terrain is away, the lower the level of detail used. As the camera approaches a piece of terrain, the level of detail is increased until the terrain is at the highest level of detail, see Figure 42.

There are two problems with geometry mip-mapping. One problem is the issue of seams between each of the terrain pieces. When one piece of terrain is at a different level of detail a crack can be generated. The crack is caused by a difference in the height between levels of detail, see figure 39. The other problem is that the transition between levels of detail generates a popping effect as the terrain transitions between levels of detail [23].

The crack effect is caused by the T-junction that is formed, see figure 40. The T-junction can be fixed by manipulating the order of the indexes. This manipulation of the vertices is a process that removes the vertex at the T-Junction, see figure 41. To adjust the vertices, offsets are specified for each side of a piece of terrain. Each piece of terrain is then given a terrain state, which identifies the levels of detail each terrain neighbor is at. If a piece of terrain is neighbored by pieces at lower levels of detail along the top and right edges, the piece of terrain is in a state where it is stitching top right. The indices are adjusted to exclude vertices along those edges, see figure 42 for results.

The popping effect is more difficult to fix. The popping side-effect can only be mitigated and not completely fixed. This effect can be mitigated through a process called morphing, which is discussed in detail in the next section.
Morphing is necessary to mitigate the popping effect associated with geometry mip-mapping. Popping is caused by the increase or decreases in geometrical information, see figure 43.

Morphing is accomplished by defining additional vertex data that stores the level of detail a vertex is removed at and the morphed position of that vertex. The morphed position is the height of the terrain at the average height of the vertices around the vertex that was removed. When a level of detail is about to switch, a linear interpolation between the full height value and the morphed value can be performed. Once every vertex in the piece of terrain is at the morphed height, the piece of terrain at the high level of detail is identical to the piece of terrain at the level of detail below it. This allows for the lower level of detail to be swapped in without the drastic change from the high level of detail to the lower level of detail, see Figure 44 [20].

REFERENCES