Sean O’Hagan

GyroScope Project Documentation

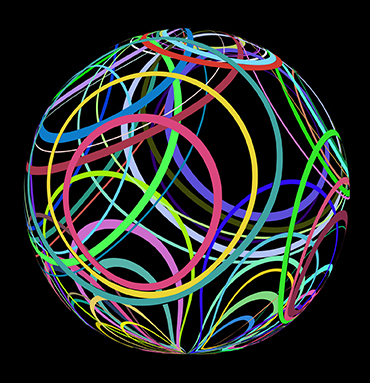


Image Designed by Kjpargeter / Freepik

# Executive Summary

This project provides a sandbox environment where end-users, through a series of menus, can construct 3D scenes composed of cubes illuminated by a real-time light source.

Users have full 3D camera control, and can travel within the scene as desired. Objects within the scene can be created, selected, deleted, moved, rotated, scaled and colored. More technically-inclined users can directly edit the shader code (the GLSL programs that get compiled by the application and executed on the GPU) and hot-reload it in the live environment to experiment with different shader effects. Objects can be instanced up to a predefined, platform-specific limit that is easily tailorable to the capabilities of the target hardware for users who choose to build from source.

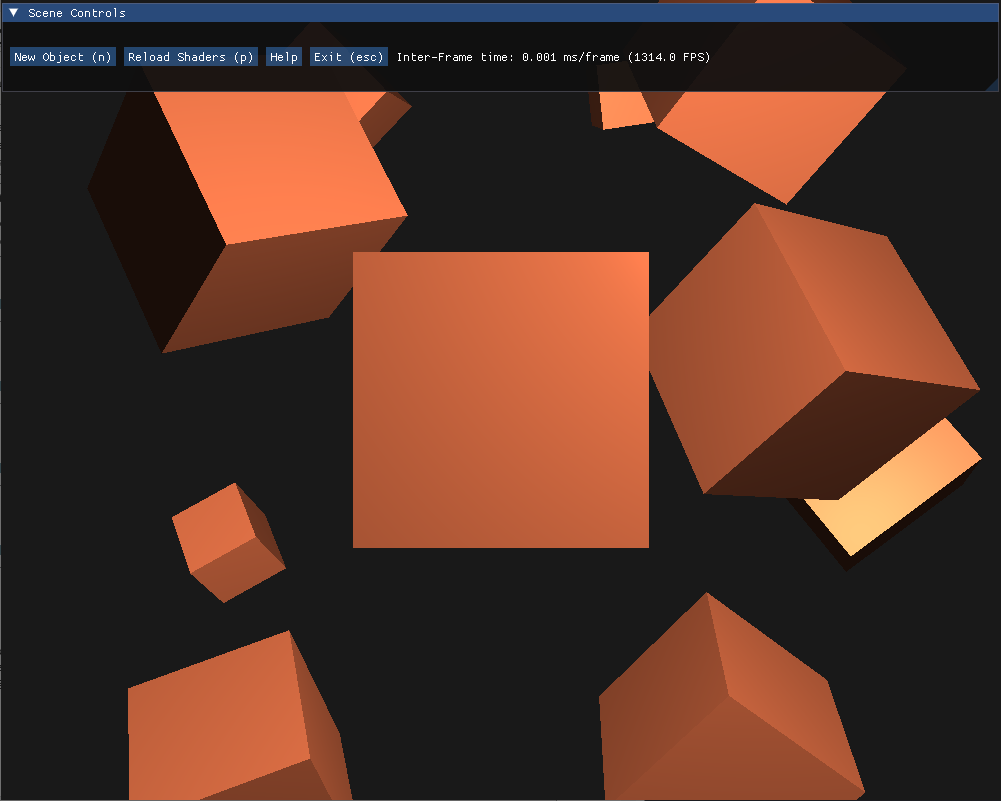
The application performs one dynamic allocation from the OS, at initialization, then manages the entirety of the sandbox’s object state through a custom, cache-friendly allocator for the entirety of the program’s lifetime. The math and physics libraries necessary to enable the 3D rendering techniques used in the application were hand-implemented – external libraries provide basic cross-platform windowing, OpenGL function pointer bookkeeping, and a basic GUI framework.

At its core, the application was designed to flexibly and easily support the addition of more advanced 3D techniques down the road. Though these capabilities were beyond the scope of this project, state memory and abstractions have been designed in a manner that will support batched multi-pass rendering, frustum culled quadtrees, clustered forward rendering, and multi-threaded dynamic physics simulations without requiring architectural changes to the base-layer implementation and memory structures.

The driving design philosophy throughout the codebase was to use direct, procedural methods to the greatest extent possible. In principle, the application ought to produce complex results through the use of simple, elegant techniques that are easy to grasp and work with, even on first contact.

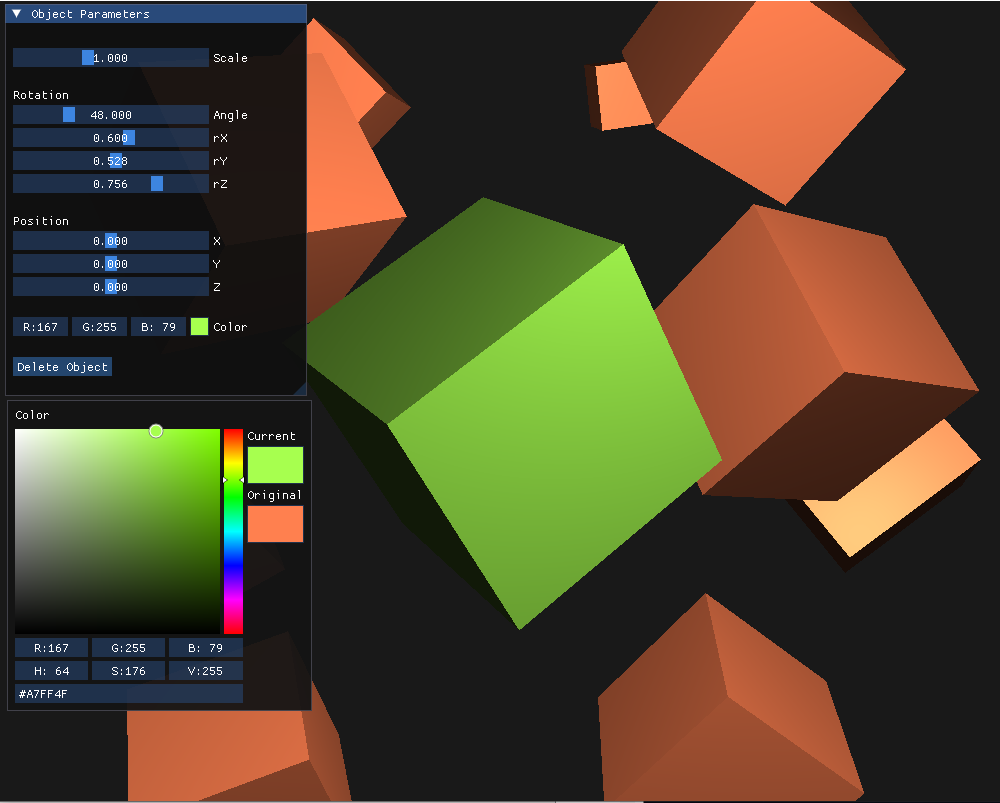
# Scenario | Default View

This is the default menu displayed to an end-user, with several objects instanced to demonstrate what an in-use system will look like. This menu appears on startup, and is returned to when any of the other menus are closed. The lighting is calculated in real-time, as objects and the camera view move relative to the light source



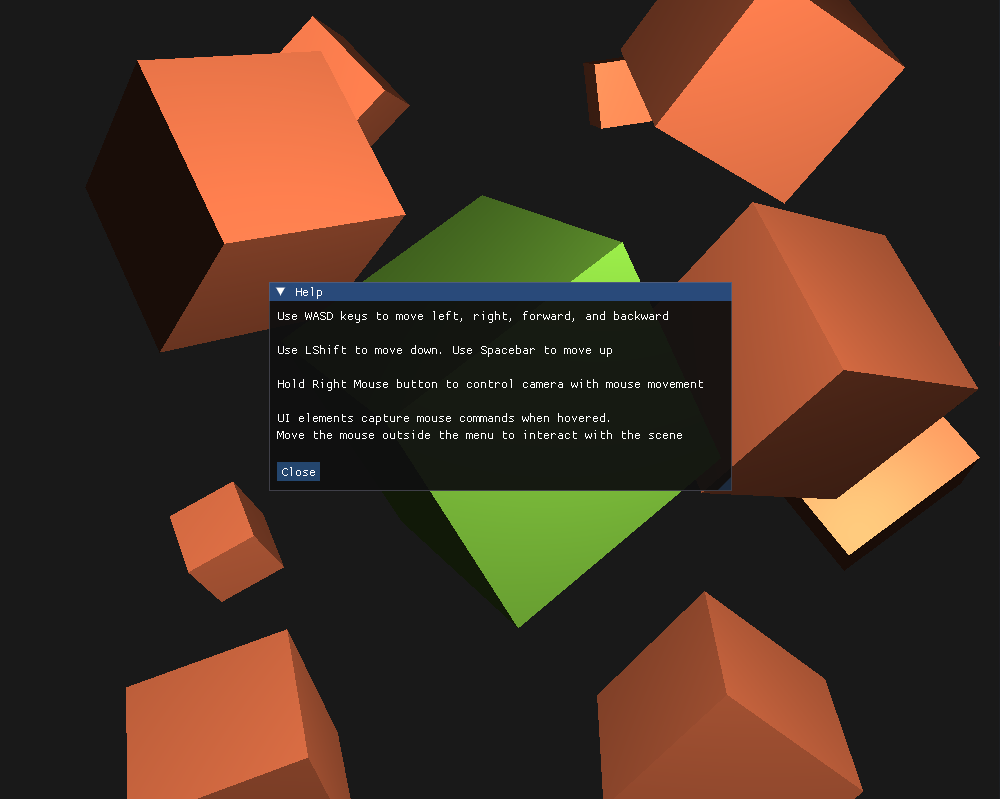
# Scenario | Object Controls

This is the interface users see when they have an object selected (either by clicking on an existing object, or using the New Object button in the default menu). In this example, the center cube from the previous scenario has had its color and rotation edited by the user, and sits at an angle where it is reflecting more light.



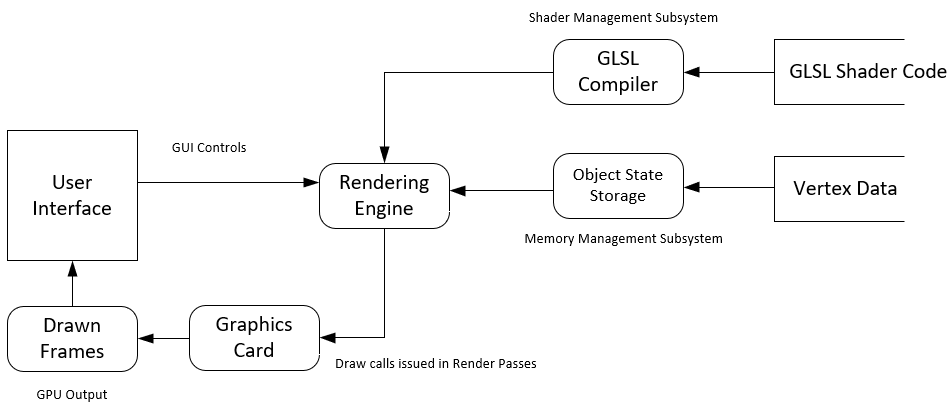
# Scenario | Help Menu

This interface provides a basic overview of the commands necessary to move the camera and interact with the scene.



# System Architecture

The system is comprised of a core rendering engine, which consumes user-input and produces commands for the various subsystems based on the received input. Subsystems receive commands from the rendering engine and produce data, which may be consumed by the core engine, other subsystems, or graphics hardware. The general layout of the architecture is described in the diagram below.



## Source Code Structure

Source code structure introduction. The following is a summary of the source code directories and their contents:

|  |  |
| --- | --- |
| **Code Directory** | |
| **Directory** | **Usage** |
| code/build | Contains the CMakeLists.txt file and is the default location for build output files, including the final executable |
| code/shaders | Shader program source files, written in GLSL. Loaded at runtime by the application |
| code/src/util | Utility modules that provide broad functionality (math functions, physics functions, memory management operations). These modules are used by application-specific routines to accomplish application-specific tasks |
| code/src | Application-specific code, including the main entry to the application |
| code/vendor/… | 3rd-party libraries used in the main application (GLFW, GLAD, DearImGUI). Each subfolder contains the necessary code to compile the library into the final executable, without requiring preinstallation by the end-user |
| *Highlighted rows indicate directories containing source code.* | |

# Executables

### GyroScope (GyroScope.exe)

Application exe. Enters via the routine in main.cpp and provides end-users with the 3D sandbox environment

# Code Architecture

The rendering engine is a set of procedural commands in main.cpp. Modules, subsystems, and hardware interfaces are initialized in several phases, then the program enters the core frame generation loop. For every frame, user input is processed, appropriate GUI elements are generated, draw calls are issued to the GPU based on data held in geometry memory, and the GPU returns processed frames to the engine, which blits them to the screen. Inter-frame upkeep data is passed from frame to frame as necessary to continue performing calculations. With the exception of a few necessary areas in subsystems, code is procedural and direct, favoring simple abstractions and functional wrappers over class hierarchies, inheritance models, and ownership flow.

External Files & Data

This application relies on a directory containing externally stored GLSL source files. These files are stored in the ‘shaders’ directory in the repository and must accompany an installation in order for it to be valid.

At runtime the shader files are opened by the application and the source code is parsed into memory. Data validation occurs, ensuring both that the file exists as a readable entity, and that the contents of the file constitute a valid set of GLSL instructions that can be successfully compiled and linked to (GPU) machine code.

If a non-existent or invalid file is supplied, the operation will fail, return an error to the user, and terminate the program. If the file is valid but the contents do not constitute a valid GLSL program, the operation will fail and return an error to the user without terminating the program. If this occurs, shader code can be edited and reloaded in the live environment until a valid program is constructed. Control over GPU memory buffers, state, and commands are not visible to the end-user – these shader files are called as part of the default rendering pipeline and do not have control over or access to system resources.

Programming Languages | C/C++ & GLSL

The core application is written entirely in C++. A few vendor libraries (GLFW, GLAD) are written in C, the rest are C++. The application also performs live compilation of user-defined shader programs, and runs these on the GPU. Shader programs are written in GLSL (OpenGL Shader Language).

Project Structures

These structures are used to abstract functionality and serve capabilities to the main application. Many make internal use of the modules described in the following section:

### **window\_handler\_t | window.cpp/.h**

This structure serves as a monolithic container of the structures and functions the application requires at runtime. The monolithic design is a requirement due to the way the GLFW library implements its callbacks, which only support a single data pointer *per application window*. This style is not my preference, and a future upgrade to this project would be to write my own cross-platform windowing base layer, replacing GLFW entirely, that provides a more typical callback interface (one data pointer *per callback).*

Members:

int Width;

int Height;

float DeltaTime;

float PrevFrameTime;

uint16\_t FirstCameraMove;

uint16\_t EditorMode;

bool ActiveSelection;

bool ReloadShaders;

bool ShouldExit;

double PrevMouseX;

double PrevMouseY;

shader\_t MainShader;

shader\_t PickShader;

fb\_mpick\_t PickPass;

mbox\_camera\_t Camera;

uMATH::mat4f\_t View;

uMATH::mat4f\_t Projection;

geometry\_create\_info\_t Active;

geometry\_state\_t GeometryObjects;

ImGuiIO ImIO;

Where basic type variables are state-savers and complex type variables are further structures and modules discussed throughout the following sections

Methods:

window\_handler\_t\* InitWindowHandler(float ScreenX, float ScreenY);

### **geometry\_state\_t | u\_mem.cpp/.h**

This structure stores contiguous, cache-friendly arrays of object state. In essence it is a 3NF data table from which the state of a particular object (keyed to its index in the arrays) can be accessed as a vertical slice through the set of arrays. Alternatively, the state of a particular characteristic for the entire set of objects (ex. the position of every single object currently in existence) can be accessed sequentially in a contiguous array.

Rather than allocating and storing many instances of individual objects with calls to OS memory, a single allocation (at initialization) acquires enough memory for the entire “table”, and manages the data through an opaque structure that provides support for custom alloc() and free() operations.

Members:

uint8\_t Visible[PROGRAM\_MAX\_OBJECTS];

float Scale[PROGRAM\_MAX\_OBJECTS];

float Intensity[PROGRAM\_MAX\_OBJECTS];

uMATH::vec3f\_t Color[PROGRAM\_MAX\_OBJECTS];

uMATH::mat4f\_t Model[PROGRAM\_MAX\_OBJECTS];

uint8\_t Position;

Private Members:

free\_list\_t FreeList;

Methods:

void Alloc();

void Alloc(const geometry\_create\_info\_t &CreateInfo);

void Free(uint8\_t FreedIndex);

### **free\_list\_t | u\_mem.ccp/.h**

This structure is a FILO stack that stores open positions in the table implemented by geometry\_state\_t. It should never be user-accessible, and is queried internally by geometry\_state\_t to provide support for cache-line-preserving free() and alloc() operations.

Members:

uint8\_t NextFreePosition = 0;

uint8\_t OpenPositions[PROGRAM\_MAX\_OBJECTS];

Methods:

void Push(uint8\_t Index);

uint8\_t Pop();

### **geometry\_create\_info\_t | u\_mem.ccp/.h**

This structure is a container that represents the state of one single object. Information can be pulled from a vertical slice of the table implemented in geometry\_state\_t into this container, based on the index of the object being looked up. A filled container can also be provided back to geometry\_state\_t in the overridden alloc() method to insert a record of the object represented by the instance of geometry\_create\_info\_t into the table implemented by geometry\_state\_t.

This object also collects procedures implemented in the uMATH library into methods that decompose a transformed 4x4 Affine matrix into its component parts (scale, rotation, position) and re-compose a new 4x4 Affine matrix based on its stored parameters. These methods are the way an object is translated to end-user GUI controls, and allow changes to a specific object in the GUI to be propagated to the table implemented by geometry\_state\_t

Members:

bool New;

bool Deleted;

float Intensity;

float Scale;

float RotationAngle;

uMATH::vec3f\_t RotationAxis;

uMATH::vec3f\_t Position;

uMATH::vec3f\_t Color;

uMATH::mat4f\_t Model;

Methods:

void DecomposeModelM4();

void ComposeModelM4();

### **mbox\_camera\_t | camera.ccp/.h**

This structure stores the state of the end-user’s current camera view frustum (prior to the perspective transform), so that it may be updated during input processing and passed as a 4x4 Affine matrix to the graphics card for calculations.

Members:

float Sensitivity = 0.1f;

float Speed = 0.0f;

float Yaw = -90.0f;

float Pitch = 0.0f;

uMATH::vec3f\_t Eye = {0.0f,0.0f,-1.0f};

uMATH::vec3f\_t UpAxis = {0.0f,1.0f,0.0f};

uMATH::vec3f\_t Position = {0.0f, 0.0f, 3.0f};

uMATH::vec3f\_t RelativeXAxis = {};

uMATH::vec3f\_t RelativeYAxis = {};

Methods:

void LookAtMouse(double XOffset, double YOffset);

void Move(GLFWwindow \*Window);

### **shader\_t | shader.ccp/.h**

This structure abstracts the code necessary to manage and verify GLSL shader programs, stored on disk, prior to their compilation into GPU memory. When changes are made to an already-compiled shader program, as long as those changes do not move or rename the source files, the shader program can be hot-reloaded in the live environment using Rebuild(). If source file names/locations have changed, Create() must be called instead with updated parameters.

Members:

uint32\_t ID;

char InfoLog[SHADER\_INFOLOG\_SIZE];

char VertPath[TMAX\_PATH\_LEN];

char FragPath[TMAX\_PATH\_LEN];

Methods:

uint32\_t Build(const char \*FilePath, int ShaderType);

int Create(const char \*VertPath, const char \*FragPath);

int Rebuild();

void Use();

### **fb\_mpick\_t | picking.ccp/.h**

This structure manages the framebuffer (GPU-side memory buffers that are not part of the main rendering pipeline and can be used to perform arbitrary GPU-accelerated computations) related to mouse picking (selecting the correct object when it’s clicked on). This framebuffer uses a depth pass to perform dynamic culling of a 2 color-channel texture that stores each object’s index in geometry\_state\_t and its type.

GetInfo() reads the pixel color of the framebuffer at the mouse’s location and returns a struct with the texel (texture pixel) data of that position. If an existing object was under the mouse, the texel data will contain its decomposed index and type. If no object exists, the texel data will return 0 in the type field.

Members:

uint32\_t FBO;

uint32\_t IndexTex;

uint32\_t DepthTex;

texel\_info\_t Info;

Methods:

int Init(uint32\_t WindowWidth, uint32\_t WindowHeight);

void Release();

void Bind\_W();

void Unbind\_W();

texel\_info\_t GetInfo(uint32\_t X, uint32\_t Y);

### **texel\_info\_t | picking.ccp/.h**

This structure is a container that stores the texel state of the picking framebuffer at the point read in from the GPU at the mouse’s coordinates.

Members:

float ID;

float Type;

### **ray\_info\_t | u\_phys.h**

This structure is part of the uPHYS namespace and stores the direction and origin of a unit vector ray cast using the uPHYS module.

Members:

uMATH::vec3f\_t Direction;

uMATH::vec3f\_t Start;

### **vec3f\_t | u\_math.h**

This structure is part of the uMATH namespace and represents a 3-dimensional vector of floating point numbers. The +, -, +=, and -= operators are overridden for operations between two 3-vectors, as these operations have trivial mathematical implementations.

Members:

float x, y, z;

### **vec4f\_t | u\_math.h**

This structure is part of the uMATH namespace and represents a 4-dimensional vector of floating point numbers. There are no overloaded operators for these structures, as they are rarely used in the application.

Members:

float x, y, z, w;

### **mat4f\_t | u\_math.h**

This structure represents a row-major 4x4 matrix as a 2-dimensional array. Operations must be ordered in row-major format (left to right) – column-major order of operations (right to left) will not produce results in line with expectations. The \* and \*= operators are overloaded for operations between two 4x4 matrixes.

Members:

float m[4][4];

Project Modules

These modules provide procedural implementations used by structures and other application code

### **uPHYS | u\_phys.h**

This module provides calculations for collision detections between a 3-dimensional ray cast from a given location and a set of objects represented as 4x4 Affine matrices in world space. If a ray is being cast for the first time, or if its position or direction should be updated, CastWorldRay() is called first, and its vec3f\_t output is passed to CheckRayOBBCollision() as the Direction parameter.

Also in CheckRayOBBCollision(), Origin is the desired ray origin (typically the position of the camera frustum), MinAABB and MaxAABB are the min and max axis-aligned bounding boxes, and Model is the 4x4 Affine matrix representing the object to be tested’s transformation in world space

Functions:

uMATH::vec3f\_t CastWorldRay(float MouseX, float MouseY, const window\_handler\_t& Window)

bool CheckRayOBBCollision(uMATH::vec3f\_t Origin, uMATH::vec3f\_t Direction,

uMATH::vec3f\_t MinAABB, uMATH::vec3f\_t MaxAABB,

uMATH::mat4f\_t Model, float\* Distance)

### **uMATH | u\_math.h**

This module implements all the linear algebra calculations necessary to perform the 3D rendering required by the main application. In the following documentation, each function declaration will be followed by a brief description.

Functions:

vec3f\_t Scalar(const vec3f\_t &v, float s)

This function non-destructively scales a 3-dimensional vector by a single floating point number.

vec4f\_t Scalar(const vec4f\_t &v, float s)

This function non-destructively scales a 4-dimensional vector by a single floating point number.

float Dot(const vec3f\_t& v, const vec3f\_t& s)

This function non-destructively returns the dot product of two 3-dimensional vectors.

vec3f\_t Cross(const vec3f\_t& v, const vec3f\_t& s)

This function non-destructively returns the cross product of two 3-dimensional vectors.

vec3f\_t Normalize(const vec3f\_t &v /\*, int\* res\*/)

This function non-destructively returns a normalized 3-dimensional vector. It returns the Z-basis vector when normalization fails (when the input vector’s length is approximately equal to 0)

void SetTransform(mat4f\_t \*t)

This function destructively sets the input matrix to an identity matrix( 1 on the diagonal, 0 everywhere else).

void SetCameraView(mat4f\_t\* t, const vec3f\_t& position, const vec3f\_t& target, const vec3f\_t& upAxis)

This function destructively sets the input matrix to a View matrix, composed from the camera object described by position, target, and upAxis. This view matrix is used to multiply an object into world space from model space – the parameters typically come from the saved camera state in window\_handler\_t

void SetFrustumHFOV(mat4f\_t \*t, float fov, float aratio, float near, float far)

This function destructively sets the input matrix to a Perspective matrix, composed from the horizontal, non-infinite perspective frustum described by fov, aratio, near, and far. This perspective matrix is used to multiply an object into screen space from world space – the parameters typically come from the saved perspective state in window\_handler\_t

void EulerRotate(mat4f\_t \*t, float theta, int axis)

This function destructively rotates an input matrix using Euler angle rotation (yaw, pitch, and roll). It is cheaper than matrix rotation, but susceptible to gimbal lock, and should not be used without corresponding limits on the input angle, theta. The rotation angle is provided in degrees.

void MatrixRotate(mat4f\_t\* t, float d, const vec3f\_t& r)

This function destructively rotates an input matrix using intra-matrix operations by an angle, d, on the complex plane specified by r, where the x, y, and z components of r represent the amount on each canonical axis the plane should be transformed by d. It is not susceptible to gimbal lock, but is much more expensive that Euler angle rotation. The rotation angle is provided in degrees and internally converted to radians.

void ExtractRotationM4(const mat4f\_t& t, float\* theta, vec3f\_t\* axis)

This function non-destructively extracts the matrix-rotation components (angle theta and plane axis) from a pure rotation matrix (a matrix that has had its scale component normalized to 1 along the diagonal, if such a scale != to 1 exists)

void Scale(mat4f\_t \*t, float s)

This function destructively performs a uniform scaling operation on the 3x3 diagonal a 4x4 Affine matrix. Non uniform scaling is not supported.

void Translate(mat4f\_t \*t, const vec3f\_t &s)

This function destructively performs a translation operation on a 4x4 Affine matrix.

vec4f\_t MultiplyV4\_M4(const vec4f\_t &v, const mat4f\_t &m)

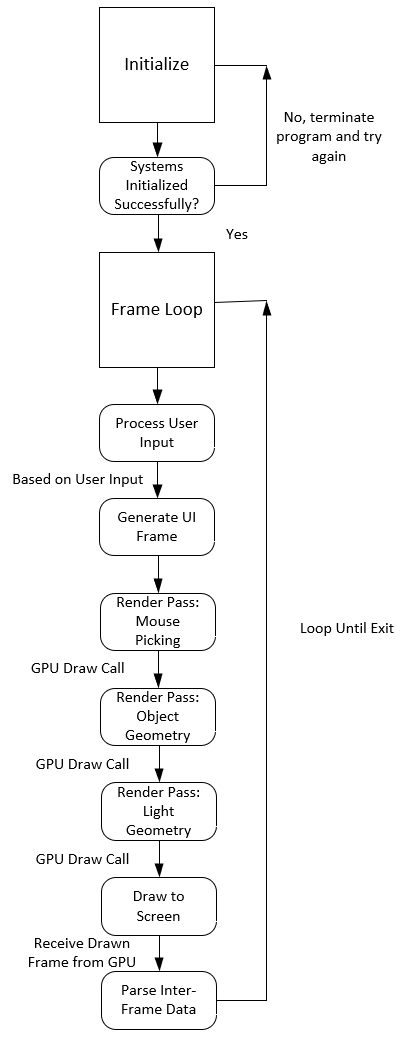
This function non-destructively multiplies a 4-dimensional vector by a 4x4 matrix.

mat4f\_t InverseM4(const mat4f\_t& m)

This function non-destructively inverses a 4x4 matrix. In cases where inversion would fail (when the determinant is approximately 0, typically occurs when attempting to inverse a non-Affine matrix), the identity matrix is returned. The returned matrix should be checked to ensure the identity matrix is not returned before the results are used.

Program Start and End Flow

This diagram below represents the program flow from initialization to exit.



Summary

This documentation lays out the functionality, system and code architecture, data structures, modules, and execution flow of the application. For further detail on specific implementation choices, review the source code in the repository.

The following appendixes (A-C) describe the release, client installation, and source build processes, respectively.

# APPENDIX A (BUILD AND RELEASE PROCESS)

This project uses a rolling-release model to provide updates and versioned releases. Patches and sub-versions will occur in-place in the repository, with changes documented in patch/version notes - these may be pulled or ignored by end-users as desired. As of now, only the current major version will receive support - users may maintain their own version of previous major releases if so desired.

For end-users on operating systems supported in the repository’s pre-compiled downloads options, new releases can be obtained by simply downloading the new release and overwriting the previous installation.

For users that build from source, the process is similar – to acquire an update, download the repo, overwriting the previous installation, then follow the instructions in Appendix D to rebuild.

# APPENDIX B (CLIENT INSTALLATION INSTRUCTIONS)

The application is statically linked, so if your desired operating system is one of the versions supported in the repository’s pre-compiled downloads options, you can download the zip file and run the application out of the (extracted) archive.

For operating systems that are not supported as a pre-compiled download, see Appendix D for instructions on building from source.

# APPENDIX C (DEVELOPER SETUP INSTRUCTIONS)

On Linux, the only requirements to set up the development environment are CMAKE, an OpenGL library (-lGL) that supports OpenGL Core Profile 4.6, and the source code from the repository.

For distros that ship feature complete, like Ubuntu or Red Hat, an up-to-date OpenGL library will be installed by default as part of the driver installation process for your detected graphics hardware. For customizable distros, like Arch or Debian, most end users will typically have these drivers (nvidia-utils, mesa-utils, etc.) installed as well. In the event that such a driver does not exist on a system, users must first install the appropriate ‘…-utils’ package for their hardware. Some distros, like Arch also provide OpenGL as a standalone package that can be installed independently of hardware drivers. Regardless of installation method, -lGL must be accessible on the machine’s PATH (typically this occurs by default alongside installation).

On Windows, OpenGL and its accompanying development tools ship as part of the default installation. The only requirements on Windows are CMAKE and an accompanying compiler. Visual Studio provides integrated support for CMAKE builds, but any compiler toolchain that CMAKE can access should work.

When building from source on either platform, note that the final executable expects access to a directory called “shaders” that contains the shader files (.frag/.vert) from the repo (out of the directory with the same name). If the application cannot find such a directory, the end-user will be prompted to provide the path to the directory themselves in the GUI – this can be anywhere on the system that the application has access to, but it must contain all shader files from the repo. Examine a pre-compiled download for an example of the standard directory layout.