СОДЕРЖАНИЕ

ВВЕДЕНИЕ 6
1 АНАЛИЗ ПРОТОТИПОВ, ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ И ФОРМИРОВАНИЕ ТРЕБОВАНИЙ К ПРОЕКТИРУЕМОМУ ПРОГРАММНОМУ СРЕДСТВУ 7
1.1 Анализ существующих аналогов 7
1.1.1 Программа «Paint 3D» 7
1.1.2 Программа «Blender» 8
1.1.3 Программа «3DS Max» 10
1.2 Постановка задачи 11
2. МОДЕЛИРОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ И РАЗРАБОТКА ФУНКЦИОНАЛЬНЫХ ТРЕБОВАНИЙ 12
2.1 Описание алгоритмов рендеринга 12
2.1.1 Алгоритм умножения матрицы на матрицу 12
2.1.2 Алгоритм умножения вектора на вектор 12
2.1.2 Алгоритм построения матрицы проекции 13
2.1.3 Алгоритм построения матрицы поворота 13
2.1.4 Алгоритм построения матриц пространства в мировых и локальных координатах 14
2.1.5 Алгоритм раскраски “Scanline” 15
2.1.6 Алгоритм отрисовки “Painter’s algorithm” 16
3. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА 18
3.1 Класс Engine3D 18
3.1.1 Конструктор класса 18
3.1.2 Деструктор класса 18
3.1.3 Класс enum class KEYBOARD_CONTROL_TYPES 18
3.1.4 Класс enum class RENDERING_STYLES 18
3.1.5 Класс enum class GE_OBJECT_TYPE 18
3.1.6 Класс enum class GE_MESH_SIDE_TYPE 19
3.1.7 Класс struct GE_STD_OBJECT_TYPES 19
3.1.8 Структура struct GE_Color 19
3.1.9 Структура struct Colors 19
3.1.10 Структура struct Triangle 21
3.1.11 Структура struct Triangle2D 21
3.1.12 Структура struct Mesh 21
3.1.13 Структура struct Mesh_Side 21
3.1.14 Структура struct GE_Object 21
3.1.15 Структура struct DrawList 23
3.1.16 Структура struct GE_Camera 23
3.1.17 Метод void DrawTopFlatTriangle() 24
3.1.18 Метод void DrawBottomFlatTriangle() 24
3.1.19 Метод void DrawFilledTriangle2D() 25
3.1.20 Метод void DrawSceneObjects() 26
3.1.21 Метод GE_Object* getGEObjectPointerByPos() 30

3.1.22 Метод void ShowPreviouslyUnneededSidesByObj() 30
3.1.23 Метод void HideUnneededSidesByObj() 32
3.1.24 Метод void CreateBlockAtSelectorPosition() 33
3.1.25 Метод void RemoveBlockAtSelectorPosition() 33
3.1.26 Метод void ChangeBlockColorAtSelectorPosition() 34
3.1.27 Метод void StartRenderLoop() 34
3.1.28 Метод void CreateCubicFormByTopMesh() 36
3.1.29. Метод void startScene() 38
4 ТЕСТИРОВАНИЕ, ПРОВЕРКА РАБОТОСПОСОБНОСТИ И АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ 39
5 РУКОВОДСТВО ПО УСТАНОВКЕ И ИСПОЛЬЗОВАНИЮ 41
5.1 Основные требования для запуска данного программного средства: 41
5.2 Руководство по установке 41
5.3 Руководство по использованию 41
ЗАКЛЮЧЕНИЕ 43
СПИСОК ЛИТЕРАТУРЫ 44
ПРИЛОЖЕНИЯ 45

ВВЕДЕНИЕ

Графический редактор – программа (или пакет программ), позволяющая создавать, просматривать, обрабатывать и редактировать цифровые изображения на компьютере. Графический редактор может быть нацелен как на плоские изображения, так и на объемные плоские.

Трёхмерная графика — раздел компьютерной графики, посвящённый методам создания изображений или видео путём моделирования объёмных объектов в трёхмерном пространстве.

Графическое изображение трёхмерных объектов отличается тем, что включает построение геометрической проекции трёхмерной модели сцены на плоскость (например, экран компьютера) с помощью специализированных программ.

Визуальные эффекты, приемы совмещения компьютерной графики с реальным видео поражают воображение и вызывают у многих интерес к трехмерному моделированию и анимации.

Трёхмерная графика активно применяется для создания изображений на плоскости экрана или листа печатной продукции в науке и промышленности, например, в системах автоматизации проектных работ (САПР; для создания твердотельных элементов: зданий, деталей машин, механизмов), архитектурной визуализации (сюда относится и так называемая «виртуальная археология»), в современных системах медицинской визуализации.

Самое широкое применение — во многих современных компьютерных играх, а также как элемент кинематографа, телевидения, печатной продукции.

Трёхмерная графика обычно имеет дело с виртуальным, воображаемым трёхмерным пространством, которое отображается на плоской, двухмерной поверхности дисплея или листа бумаги. В настоящее время известно несколько способов отображения трёхмерной информации в объёмном виде, хотя большинство из них представляет объёмные характеристики весьма условно, поскольку работают со стереоизображением. Из этой области можно отметить стереоочки, виртуальные шлемы, 3D-дисплеи, способные демонстрировать трёхмерное изображение.

1 АНАЛИЗ ПРОТОТИПОВ, ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ И ФОРМИРОВАНИЕ ТРЕБОВАНИЙ К ПРОЕКТИРУЕМОМУ ПРОГРАММНОМУ СРЕДСТВУ

1.1 Анализ существующих аналогов

1.1.1 Программа «Paint 3D»[1]

Pain 3D — многофункциональный, но в то же время довольно простой в использовании 3D графический редактор компании Microsoft, входящий в состав всех операционных систем Windows, начиная с Windows 8.

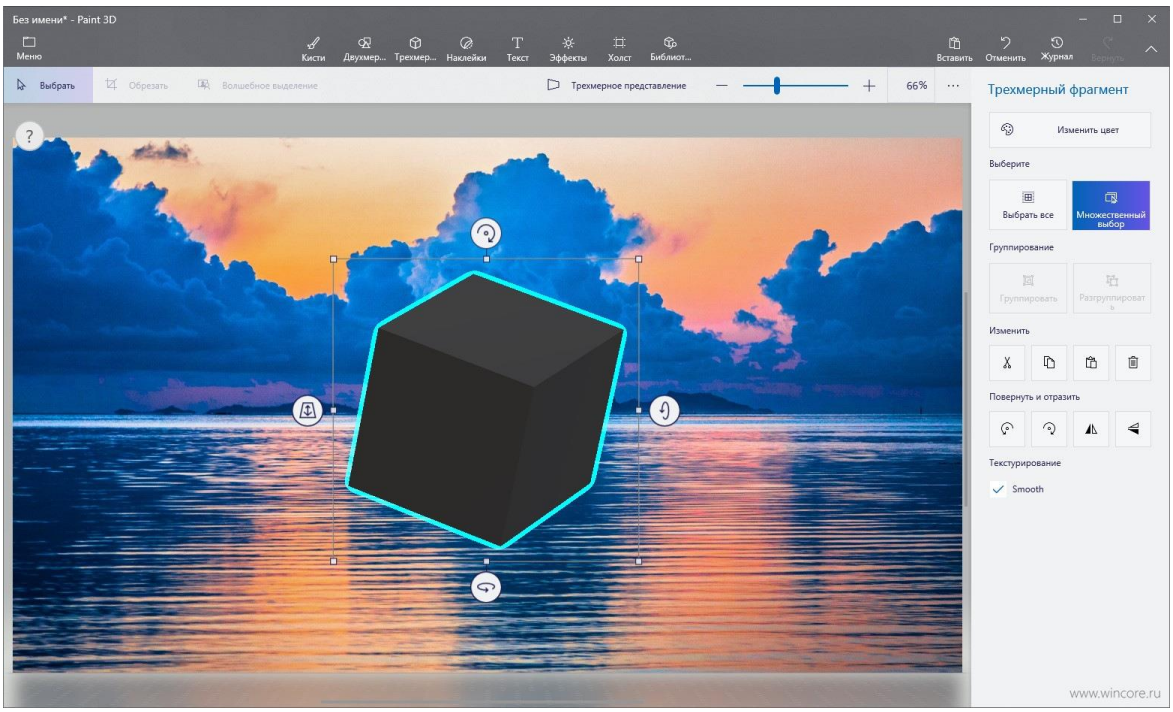


Рисунок 1.1 – Интерфейс графического редактора Paint 3D

Данный графический редактор обладает рядом таких возможностей, как:
1) Многократное увеличение или уменьшение фигуры;
2) Выбор цвета для рисования;
3) Пипетка;
4) Заливка;
5) Замена цвета или ластик;
6) Выделение;
7) Подрезка фигур;
Несмотря на это, у редактора Paint 3D существуют свои недостатки. Среди них можно выделить:
1) Отсутствие возможности залить градиентом (можно сделать только с фигурами);
2) Отсутствие поддержки большинства форматов 3D файлов;
3) Отсутствие поддержки прозрачности;
4) Отсутствие слоёв.

1.1.2 Программа «Blender»[2]

Blender — профессиональное свободное и открытое программное обеспечение для создания трёхмерной компьютерной графики, включающее в себя средства моделирования, скульптинга, анимации, симуляции, рендеринга, постобработки и монтажа видео со звуком, компоновки с помощью «узлов» (Node Compositing), а также создания 2D-анимаций. В настоящее время пользуется большой популярностью среди бесплатных 3D-редакторов в связи с его быстрым стабильным развитием и технической поддержкой.

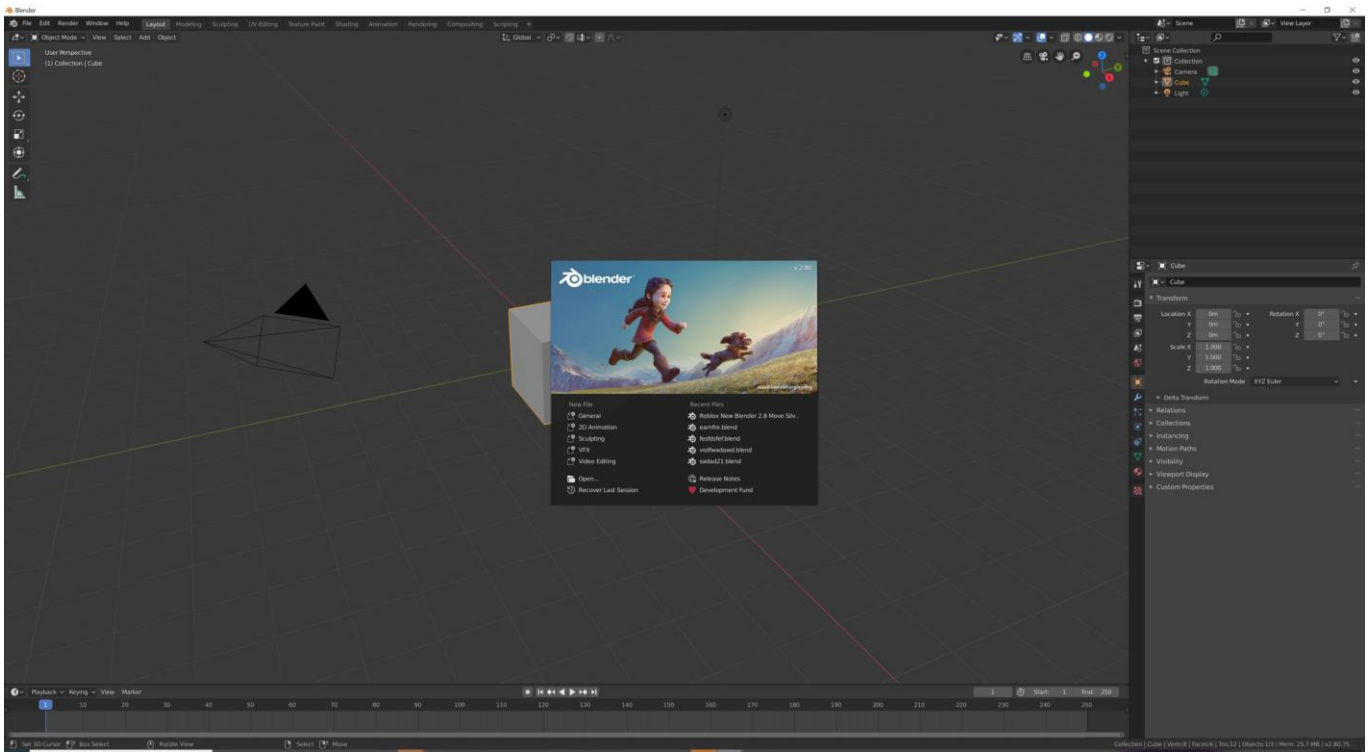


Рисунок 1.2 – Интерфейс графического редактора Blender

Рисунок 1.3 – Пример сцены, созданной в Blender

Функции:

1) Поддержка разнообразных геометрических примитивов, включая полигональные модели, систему быстрого моделирования в режиме subdivision surface (SubSurf), кривые Безье, поверхности NURBS, metaballs (метасферы), скульптурное моделирование и векторные шрифты.
2) Универсальные встроенные механизмы рендеринга и интеграция с внешними рендерерами YafRay, LuxRender и многими другими.
3) Инструменты анимации, среди которых инверсная кинематика, скелетная анимация и сеточная деформация, анимация по ключевым кадрам, нелинейная анимация, редактирование весовых коэффициентов вершин, ограничители.
4) Динамика мягких тел (включая определение коллизий объектов при взаимодействии), динамика твёрдых тел на основе физического движка Bullet.
5) Система частиц включающая в себя систему волос на основе частиц.
6) Модификаторы для применения неразрушающих эффектов.
7) Язык программирования Python используется как средство определения интерфейса, создания инструментов и прототипов, системы логики в играх, как средство импорта/экспорта файлов (например, COLLADA), автоматизации задач.
8) Базовые функции нелинейного видео и аудио монтажа.
9) Композитинг видео, работа с хромакеем.
10) Трекинг камеры и объектов.
11) Real-time контроль во время физической симуляции и рендеринга.
12) Процедурное и node-based текстурирование, а также возможность рисовать текстуру прямо на модели.
13) Grease Pencil — инструмент для 2D-анимации в полном 3D-пайплайне.
14) Blender Game Engine — подпроект Blender, предоставляющий интерактивные функции, такие как определение коллизий, движок динамики и программируемая логика. Также он позволяет создавать отдельные real-time-приложения начиная от архитектурной визуализации до видео игр. Удалён в версии 2.8.

В поставку Blender входят:

Blender Render (Blender Internal) — оригинальный движок рендеринга Blender, исходный код которого был написан еще в 90-х. Это смесь новых и старых технологий рендеринга, включающих в себя трассировку лучей, подповерхностное рассеивание, глянцевые отражения и даже примитивная система глобального освещения. Удалён из Blender в версии 2.8

Cycles Render — рендер без допущений, с возможностью рендеринга на GPU. Входит в поставку Blender с версии 2.61.

Clay Render — Гипсовый рендер, применяет материал глины ко всем объектам сцены, без изменения их материалов. Входит в поставку Blender с версии 2.79.

EEVEE — Назван в честь покемона Иви, позже придумана расшифровка Extra Easy Virtual Environment Engine. Представляет из себя полнофункциональный PBR движок для визуализации в реальном времени.

1.1.3 Программа «3DS Max»[3]

Autodesk 3ds Max (ранее 3D Studio MAX) — профессиональное программное обеспечение для 3D-моделирования, анимации и визуализации при создании игр и проектировании. В настоящее время разрабатывается и издается компанией Autodesk.

Программа доступна по подписке для коммерческих целей от одного месяца до трёх лет. Для студентов и преподавателей подписка на три года бесплатная, но с такой лицензией программу можно использовать только для обучения.

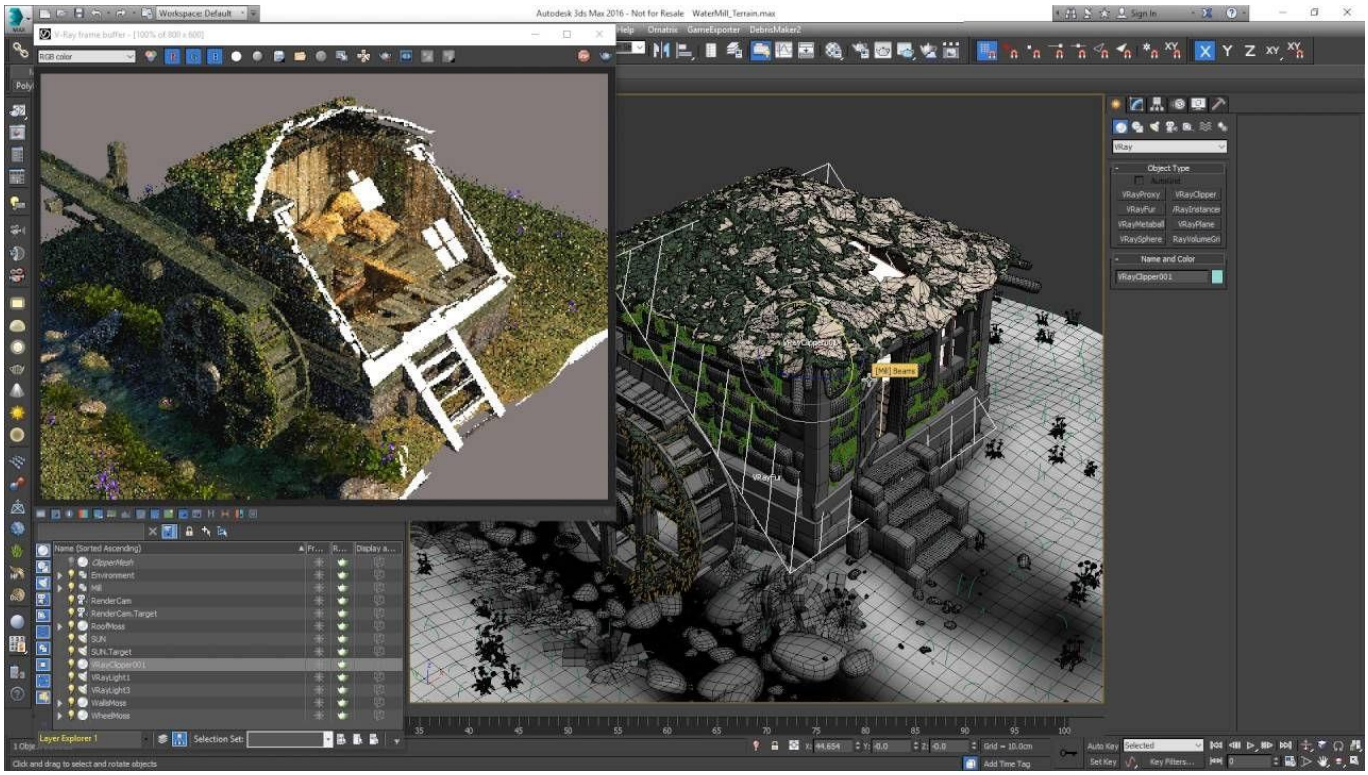


Рисунок 1.4 – Интерфейс графического редактора 3DS Max

3ds Max располагает обширными средствами для создания разнообразных по форме и сложности трёхмерных компьютерных моделей, реальных или фантастических объектов окружающего мира, с использованием разнообразных техник и механизмов, включающих следующие:

полигональное моделирование, в которое входят Editable mesh (редактируемая поверхность) и Editable poly (редактируемый полигон) — это самый распространённый метод моделирования, используется для создания сложных моделей и низкополигональных моделей для игр.

Как правило, моделирование сложных объектов с последующим преобразованием в Editable poly начинается с построения параметрического объекта «Box», и поэтому способ моделирования общепринято называется «Box modeling»;

моделирование на основе неоднородных рациональных B-сплайнов (NURBS) (следует отметить, что NURBS-моделирование в 3ds Max-е настолько примитивное, что никто этим методом практически не пользуется);

моделирование на основе т. н. «сеток кусков» или поверхностей Безье (Editable patch) — подходит для моделирования тел вращения;

моделирование с использованием встроенных библиотек стандартных параметрических объектов (примитивов) и модификаторов;

моделирование на основе сплайнов (Spline) с последующим применением модификатора Surface — примитивный аналог NURBS, удобный, однако, для создания объектов со сложными перетекающими формами, которые трудно создать методами полигонального моделирования;

моделирование на основе сплайнов с последующим применением модификаторов Extrude, Lathe, Bevel Profile или создания на основе сплайнов объектов Loft. Этот метод широко применяется для архитектурного моделирования.

Методы моделирования могут сочетаться друг с другом.

Моделирование на основе стандартных объектов, как правило, является основным методом моделирования и служит отправной точкой для создания объектов сложной структуры, что связано с использованием примитивов в сочетании друг с другом как элементарных частей составных объектов.

1.2 Постановка задачи

Так как проектируемое программное средство должно соответствовать названию «3D графический редактор», необходимо реализовать следующие функции:

1) Создание 3D объекта (куб)
2) Перемещение объекта в пространстве
3) Изменение цвета объекта
4) Реализация освещение
5) Возможность перемещать камеру
6) Возможность менять FOV
7) Возможность менять угол наклона камеры

В качестве языка программирования выбран язык C++ и кроссплатформенный фреймворк SDL2.

2. МОДЕЛИРОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ И РАЗРАБОТКА ФУНКЦИОНАЛЬНЫХ ТРЕБОВАНИЙ

2.1 Описание алгоритмов рендеринга

Для создания проекции 3D мира на 2D экран используются 4х мерные матрицы и 3х мерные векторы. А также алгоритмы векторной алгебры: умножения матрицы на матрицу, вектора на вектор, построение матриц поворота, матриц проекции, матриц пространства в мировых и локальных координатах [4]. Для раскраски полигонов используется алгоритм Scanline. Для правильной отрисовки также используется Painter's algorithm. Для создания освещения используется алгоритм сравнения угла поворота полигона с углом поворота камеры [5].

2.1.1 Алгоритм умножения матрицы на матрицу

Алгоритм представлен на рисунке ниже

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \cdot \begin{vmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{vmatrix} =$$

$$= \begin{vmatrix} a_{11}\cdot b_{11} + a_{12}\cdot b_{21} + a_{13}\cdot b_{31} & a_{11}\cdot b_{12} + a_{12}\cdot b_{22} + a_{13}\cdot b_{32} & a_{11}\cdot b_{13} + a_{12}\cdot b_{23} + a_{13}\cdot b_{33} \\ a_{21}\cdot b_{11} + a_{22}\cdot b_{21} + a_{23}\cdot b_{31} & a_{21}\cdot b_{12} + a_{22}\cdot b_{22} + a_{23}\cdot b_{32} & a_{21}\cdot b_{13} + a_{22}\cdot b_{23} + a_{23}\cdot b_{33} \\ a_{31}\cdot b_{11} + a_{32}\cdot b_{21} + a_{33}\cdot b_{31} & a_{31}\cdot b_{12} + a_{32}\cdot b_{22} + a_{33}\cdot b_{32} & a_{31}\cdot b_{13} + a_{32}\cdot b_{23} + a_{33}\cdot b_{33} \end{vmatrix}$$

Рисунок 2.1 – Алгоритм умножения матрицы на матрицу

2.1.2 Алгоритм умножения вектора на вектор

Алгоритм представлен на рисунке ниже

$$(a\ \ b\ \ c\ \ d)\cdot\begin{pmatrix} s \\ x \\ y \\ z \end{pmatrix} \rightarrow a\cdot s + b\cdot x + c\cdot y + d\cdot z$$

$$\begin{pmatrix} s \\ x \\ y \\ z \end{pmatrix}\cdot(a\ \ b\ \ c\ \ d) \rightarrow \begin{pmatrix} a\cdot s & s\cdot b & s\cdot c & s\cdot d \\ a\cdot x & b\cdot x & x\cdot c & d\cdot x \\ y\cdot a & b\cdot y & c\cdot y & y\cdot d \\ z\cdot a & z\cdot b & c\cdot z & d\cdot z \end{pmatrix}$$

Рисунок 2.2 – Алгоритм умножения вектора на вектор

2.1.2 Алгоритм построения матрицы проекции

Матрица проекций отвечает за то, какой объём пространства будет визуализироваться, каким образом вершины графических примитивов будут спроецированы на двумерную поверхность экрана монитора. Преобразования матрицы проекций ведут к тому, что все изображение будет изменяться (масштабироваться, перемещаться или вращаться). Возможно использование двух режимов матрицы проекций: перспективная и ортографическая.

При перспективной проекции используется тот факт, что для человеческий глаз работает с предметом дальнего типа, размеры которого имеют угловые размеры. Чем дальше объект, тем меньше он нам кажется. Таким образом, объём пространства, который визуализируется представляет собой пирамиду.

$$T_{\substack{\text{резперсп}\\ 2\text{точ}\\ C_z}} = T_{\text{вр}Y} \cdot T_{\substack{\text{перенос}\\ l_X, m_z, n_z}} \cdot T_{\substack{\text{персп}\\ 1\text{точ}\\ C_z}} \cdot T_{\substack{\text{орт}\\ Z_c=0}} \qquad (18)$$

Перемножив последовательно матрицы соотношения (18), получим требуемый результат:

$$T_{\substack{\text{резперсп}\\ 2\text{точ}\\ Z_c}} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l_X & m_Y & n_Z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & 0 & -r\sin\theta \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & 0 & r\cos\theta \\ l_X & m_Y & 0 & n_Z r+1 \end{bmatrix} \qquad (19)$$

Рисунок 2.3 – Алгоритм построения матрицы проекции

2.1.3 Алгоритм построения матрицы поворота

Ма́трицей поворо́та (или матрицей направляющих косинусов) называется ортогональная матрица, которая используется для выполнения собственного ортогонального преобразования в евклидовом пространстве. При умножении любого вектора на матрицу поворота длина вектора сохраняется. Определитель матрицы поворота равен единице.

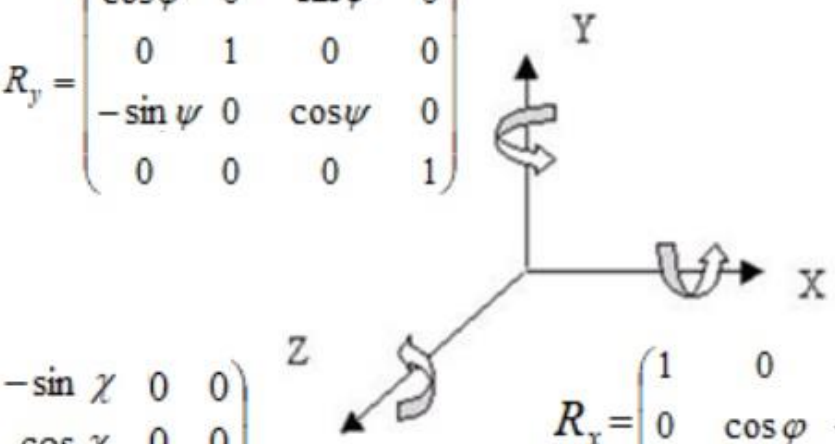
Обычно считают, что в отличие от матрицы перехода при повороте системы координат (базиса), при умножении на матрицу поворота вектора-столбца координаты вектора преобразуются в соответствии с поворотом самого вектора (а не поворотом координатных осей; то есть при этом координаты повернутого вектора получаются в той же, неподвижной системе координат). Однако отличие той и другой матрицы лишь в знаке угла поворота, и одна может быть получена из другой заменой угла поворота на противоположный; та и другая взаимно обратны и могут быть получены друг из друга транспонированием.

Любое вращение в трёхмерном пространстве может быть представлено как композиция поворотов вокруг трёх ортогональных осей (например, вокруг осей декартовых координат). Этой композиции соответствует матрица, равная произведению соответствующих трёх матриц поворота.

Матрицами вращения вокруг оси декартовой системы координат на угол в трёхмерном пространстве с неподвижной системой координат являются:

Матрицы поворота

$$R_y = \begin{pmatrix} \cos\psi & 0 & \sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$R_z = \begin{pmatrix} \cos\chi & -\sin\chi & 0 & 0 \\ \sin\chi & \cos\chi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Рисунок 2.4 – Алгоритм построения матриц поворота

2.1.4 Алгоритм построения матриц пространства в мировых и локальных координатах

Мировая матрица – позволяет производить различные матричные преобразования (трансформацию и масштабирование) объекта в мировой системе координат. Мировая система координат – это своя локальная система координат данного объекта, которой наделяется каждый объект, скажем так прошедший через мировую матрицу, поскольку каждая вершина участвует в произведении этой матрицы.

Новая локальная система координат значительно упрощает аффинные преобразования объекта в пространстве. Например, чтобы перенести объект с левого верхнего угла дисплея в нижний правый угол дисплея, то есть переместить игровой объект в пространстве, необходимо просто перенести его локальную точку отсчета, системы координат на новое место. Если бы не было мировой матрицы, то этот объект пришлось переносить по одной вершине. Поэтому любой объект, а точнее все вершины этого объекта проходят через мировую матрицу преобразования.

$$\mathbf{R} = \begin{bmatrix} n_1^2 + (1 - n_1^2)\cos\theta & n_1 n_2 (1 - \cos\theta) + n_3 \sin\theta & n_1 n_3 (1 - \cos\theta) - n_2 \sin\theta & 0 \\ n_1 n_2 (1 - \cos\theta) - n_3 \sin\theta & n_2^2 + (1 - n_2^2)\cos\theta & n_2 n_3 (1 - \cos\theta) + n_1 \sin\theta & 0 \\ n_1 n_3 (1 - \cos\theta) + n_2 \sin\theta & n_2 n_3 (1 - \cos\theta) - n_1 \sin\theta & n_3^2 + (1 - n_3^2)\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Рисунок 2.5 – Алгоритм построения мировой матрицы

2.1.5 Алгоритм раскраски “Scanline”

Метод «Scanline» — один из методов рендеринга в компьютерной графике, при котором сцена строится на основе замеров пересечения лучей с визуализируемой поверхностью.

Рейкастинг не является синонимом к рейтрейсингу (трассировке лучей), но он может быть представлен как сокращённая и существенно более быстрая версия алгоритма трассировки лучей. Оба алгоритма являются «image order» и используются в компьютерной графике для рендеринга трёхмерных сцен на двухмерный экран с помощью проекционных лучей, которые проецируются от глаз наблюдателя к источнику света. Метод бросания лучей не вычисляет новые тангенсы лучей света, которые возникнут после того, когда луч, который проецируется от глаза к источнику света, пересечётся с поверхностью. Эта особенность делает невозможным точный рендеринг отражений, преломлений и естественной проекции теней с помощью рейкастинга. Однако все эти особенности могут быть добавлены с помощью «фальшивых» (обманных, аппроксимационных) методик, например, через использование текстурных карт или другие методы. Высокая скорость вычисления сделала рейкастинг удобным методом рендеринга в ранних компьютерных играх с трёхмерной графикой реального времени.

В реальной природе источник света испускает луч света, который, «путешествуя» по пространству, в конечном счёте «натыкается» на какую-либо преграду, которая прерывает распространение этого светового луча. Луч света можно представить в виде потока фотонов, который движется вдоль вектора луча. В какой-либо точке пути с лучом света может случиться любая комбинация трёх вещей: поглощение, отражение и преломление. Поверхность может отразить весь световой луч или только его часть в одном или нескольких направлениях. Поверхность может также поглотить часть светового луча, что приводит к потере интенсивности отраженного и/или преломлённого луча. Если поверхность имеет свойства прозрачности, то она преломляет часть светового луча внутри себя и изменяет его направление распространения, поглощая некоторый (или весь) спектр луча (и, возможно, изменяя цвет). Суммарная интенсивность светового луча, которая была «потеряна» вследствие поглощения, преломления и отражения, должна быть в точности равной исходящей (начальной) интенсивности этого луча. Поверхность не может, например, отразить 66% входящего светового луча, и преломить 50%, так как сумма этих порций будет равной 116%, что больше 100%. Отсюда вытекает, что отраженные и/или преломлённые лучи должны «стыкаться» с другими поверхностями, где их поглощающие, отражающие и преломляющие способности снова вычисляются, основываясь на результатах вычислений входящих лучей. Некоторые из лучей, сгенерированных источником света, распространяются по пространству и, в конечном счете, попадают на область просмотра (глаз человека, объектив фото- или видеокамеры и т.д.). Попытка симулировать физический процесс распространения света путём трассировки световых лучей, используя компьютер, является чрезмерно расточительной, так как только незначительная доля лучей, сгенерированных источником света, попадает на область просмотра.

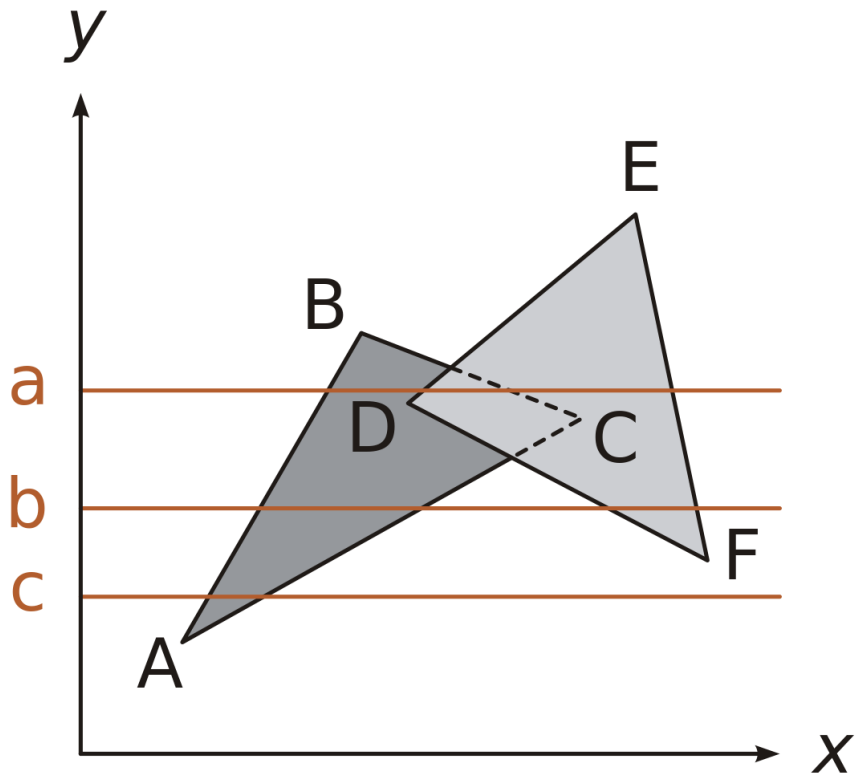


Рисунок 2.6 – Алгоритм Scanline

2.1.6 Алгоритм отрисовки “Painter’s algorithm”

Painter’s algorithm — простейший программный вариант решения «проблемы видимости» в трехмерной компьютерной графике.

Название «алгоритм художника» относится к технике, используемой многими живописцами: сначала рисуются наиболее удалённые части сцены, потом части которые ближе. Постепенно ближние части начинают перекрывать отдаленные части более удалённых объектов. Задача программиста при реализации алгоритма художника — отсортировать все полигоны по удалённости от наблюдателя и начать выводить, начиная с более дальних.

Пример работы приведен на рисунке ниже:

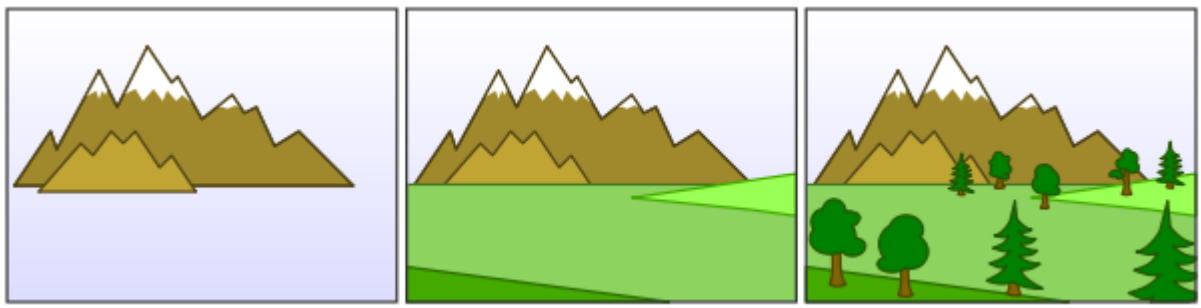


Рисунок 2.7 – Алгоритм отрисовки “Painter’s algorithm”

Наиболее удалённые горы рисуются в первую очередь. Затем рисуется луг, а потом деревья. Можно заметить, что некоторые части луга ближе, чем некоторые деревья. Однако правильный порядок сортировки (горы -> луг -> деревья) позволяет получить корректную картину, без перекрытия деревьев частями луга.

Проблемы алгоритма:

Алгоритм не позволяет получить корректную картину в случае взаимноперекрывающихся полигонов. В этом случае, как показано на рисунке справа, полигоны A, B и C накладываются друг на друга таким образом, что невозможно определить, в каком порядке их следует рисовать. В этом случае, следует разбить конфликтный полигон на несколько меньших, например алгоритмом Ньюэлла, предложенным в 1972 году.

Второй распространённой проблемой является то, что система прорисовывает также области, которые впоследствии будут перекрыты, на что тратится лишнее процессорное время.

Эти недостатки привели к разработке метода Z-буфера, который можно рассматривать как развитие алгоритма художника.

3. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

3.1 Класс Engine3D

3.1.1 Конструктор класса

Сигнатура конструктора класса Engine3D имеет следующий вид: Engine3D(int _WIDTH, int _HEIGHT)).

Основная цель конструктора – указать размер создаваемого окна. А также проинициализировать все графические библиотеки и стандартные полигональные сетки.

3.1.2 Деструктор класса

Задача деструктора – выгрузить из памяти графические библиотеки для корректного завершения работы программы.

3.1.3 Класс enum class KEYBOARD_CONTROL_TYPES

Данный класс служит для определения того, как движок должен реагировать на действия пользователя.

Описание:

```
enum class KEYBOARD_CONTROL_TYPES {
        ALLOW_SCENE_EDITING,
        ALLOW_OBJECT_EDITING,
        ALLOW_CAMERA_CONTROL
 };
```

3.1.4 Класс enum class RENDERING_STYLES

Данный класс служит для определения того, в каком стиле движок должен отрисовывать объекты.

Описание:

```
enum class RENDERING_STYLES {
        STD_SHADED,
        STD_POLY_SHADED,
        DEBUG_DRAW_ONLY_POLYGONS
 };
```

3.1.5 Класс enum class GE_OBJECT_TYPE

Данный класс служит для определения типа объекта.

Описание:

```
enum class GE_OBJECT_TYPE {
        UNDEFINED,
        SELECTOR,
        CUBE
 };
```

3.1.6 Класс enum class GE_MESH_SIDE_TYPE

Данный класс служит для определения направления нормали полигона. Указание направления нормали позволяет оптимизировать рендеринг и избавиться от проблемы алгоритма “Painter’s algorithm”.

Описание:

```
enum class GE_MESH_SIDE_TYPE {
        UNDEFINED,
        SOUTH,
        EAST,
        NORTH,
        WEST,
        TOP,
        BOTTOM
 };
```

3.1.7 Класс struct GE_STD_OBJECT_TYPES

Данная структура содержит экземпляры стандартных объектов.

Описание:

```
struct GE_STD_OBJECT_TYPES {
        GE_Object SELECTOR;
        GE_Object CUBE;
 };
```

3.1.8 Структура struct GE_Color

Данная структура содержит информацию о цвете в формате RGB.

Описание:

```
struct GE_Color {
        float R;
        float G;
        float B;
 };
```

3.1.9 Структура struct Colors

Данная структура очень упрощает работу с цветами. Заданы стандартные цвета. Также определены 2 функции, позволяющие циклически менять цвета (используется при раскраске объектов на сцене).

Описание:

```
struct Colors {
  private:
        GE_Color WHITE = { 255.0f, 255.0f, 255.0f };
        GE_Color YELLOW = { 250.0f, 211.0f, 0.0f };
        GE_Color BLUE = { 32.0f, 5.0f, 137.0f };
        GE_Color AQUA = { 0.0f, 191.0f, 255.0f };
        GE_Color VIOLET = { 138.0f, 43.0f, 226.0f };
        GE_Color PURPLE = { 223.0f, 0.0f, 254.0f };
        GE_Color GREEN = { 62.0f, 255.0f, 21.0f };
```

```
	GE_Color RED = { 254.0f, 0.0f, 0.0f };
public:
	enum class Types {
		WHITE,
		YELLOW,
		BLUE,
		AQUA,
		VIOLET,
		PURPLE,
		GREEN,
		RED
	};
	GE_Color getColorByType(Types t) {
		// Invalid type will return WHITE
		switch (t) {
			case Types::WHITE: return WHITE; break;
			case Types::YELLOW: return YELLOW; break;
			case Types::BLUE: return BLUE; break;
			case Types::AQUA: return AQUA; break;
			case Types::VIOLET: return VIOLET; break;
			case Types::PURPLE: return PURPLE; break;
			case Types::GREEN: return GREEN; break;
			case Types::RED: return RED; break;
			default: return WHITE; break;
		}
	}
	Types getColorTypeAfter(Types t) {
		// Invalid type will return WHITE
		switch (t) {
			case Types::WHITE: return Types::YELLOW; break;
			case Types::YELLOW: return Types::BLUE; break;
			case Types::BLUE: return Types::AQUA; break;
			case Types::AQUA: return Types::VIOLET; break;
			case Types::VIOLET: return Types::PURPLE; break;
			case Types::PURPLE: return Types::GREEN; break;
			case Types::GREEN: return Types::RED; break;
			case Types::RED: return Types::WHITE; break;
			default: return Types::WHITE; break;
		}
	}
};
```

3.1.10 Структура struct Triangle

Структура содержит информацию о векторах, на которых построен полигон и цвет полигона.

Описание:

```
struct Triangle {
        vec3 p[3];
        GE_Color color = { 255.0f, 255.0f, 255.0f };
 };
```

3.1.11 Структура struct Triangle2D

Структура содержит информацию о треугольнике для отрисовки на 2D экране.

Описание:

```
struct Triangle2D {
        SDL_Point p[3];
 };
```

3.1.12 Структура struct Mesh

Структура, содержащая полигональную сетку.

Описание:

```
struct Mesh {
        std::vector<Triangle> polygons; // First is always considered as a selector
 };
```

3.1.13 Структура struct Mesh_Side

Структура, позволяющая придать свойства конкретной полигональной сетке.

Описание:

```
struct Mesh_Side {
        GE_MESH_SIDE_TYPE type = GE_MESH_SIDE_TYPE::UNDEFINED;
        bool hidden = false;
        Mesh mesh;
 };
```

3.1.14 Структура struct GE_Object

Структура, содержащаю всю информацию об объекте на сцене. Также определены функции, позволяющие безопасно менять свойства конкретного объекта.

Описание:

```
struct GE_Object {
 private:
        vec3 position = { 0, 0, 0 };
        GE_OBJECT_TYPE objType = GE_OBJECT_TYPE::UNDEFINED;
        Colors::Types colorType = Colors::Types::WHITE;
 public:
        std::vector<Mesh_Side> sides;

        vec3 getPosition() {
                return position;
        }
```

```
		void setObjType(GE_OBJECT_TYPE t) {
			objType = t;
		}

		GE_OBJECT_TYPE getObjType() {
			return objType;
		}

		Colors::Types getColorType() {
			return colorType;
		}

		void moveBy(vec3 v) {
			position = Vector3_Add(position, v);
			for (Mesh_Side &side : sides) {
				for (Triangle &tri : side.mesh.polygons) {
					for (vec3 &p : tri.p) {
						p = Vector3_Add(p, v);
					}
				}
			}
			printf("Moved block to %.2f %.2f %.2f\n", position.x, position.y, position.z);
		}

		void moveTo(vec3 v) {
			vec3 offset = Vector3_Sub(v, position);
			moveBy(offset);
		}

		void scaleBy(float k) {
			for (Mesh_Side &side : sides) {
				for (Triangle &tri : side.mesh.polygons) {
					for (vec3 &p : tri.p) {
						p = Vector3_Mul(p, k);
					}
				}
			}
			printf("Scaled block by %.2f\n", k);
		}

		void setColor(GE_Color _color, Colors::Types _t) {
			colorType = _t;
```

```
                for (Mesh_Side &side : sides) {
                        for (Triangle &tri : side.mesh.polygons) {
                                tri.color = _color;
                        }
                }
                printf("Changed color to %.2f %.2f %.2f\n", _color.R, _color.G,
_color.B);
        }

        void hideSide(GE_MESH_SIDE_TYPE type) {
                for (Mesh_Side &side : sides) {
                        if (side.type == type) {
                                side.hidden = true;
                                break;
                        }
                }
        }

        void showSide(GE_MESH_SIDE_TYPE type) {
                for (Mesh_Side &side : sides) {
                        if (side.type == type) {
                                side.hidden = false;
                                break;
                        }
                }
        }
  };
```

3.1.15 Структура struct DrawList

Структура, которая содержит вектор объектов, которые требуется просчитать и вывести на экран при очередной генерации кадра.

Описание:

```
struct DrawList {
        GE_Object selectorBox;
        std::vector<GE_Object> obj;
  };
```

3.1.16 Структура struct GE_Camera

Структура, содержащая информацию о камере.

Описание:

```
struct GE_Camera {
        // pi -3.14159f
        vec3 position = { 0.0f, 0.0f, 0.0f };
        vec3 lookDirection = { 0.0f, 0.0f, 1.0f };
        float fXRotation = 0.0f;
        float fYRotation = 0.0f;
```

```
        float fFOV = 90.0f;
        float fNear = 0.1f;
        float fFar = 1000.0f;
  };
```

3.1.17 Метод void DrawTopFlatTriangle()

Метод производит отрисовку треугольника, у которого плоская часть сверху. Алгоритм заключается в том, что треугольник заполняется сверху-вниз лииями.
Описание:

```
void DrawTopFlatTriangle(SDL_Renderer *renderer, SDL_Point *v)
  {
        /*
            0----------1
              \         /
               \       /
                \     /
                 \   /
                  \ /
                   2
        */
        float dx0 = (float)(v[0].x - v[2].x) / (float)(v[2].y - v[0].y);
        float dx1 = (float)(v[1].x - v[2].x) / (float)(v[2].y - v[1].y);

        float xOffset0 = v[2].x;
        float xOffset1 = v[2].x;

        for (int scanlineY = v[2].y; scanlineY >= v[0].y; scanlineY--)
        {
              SDL_RenderDrawLine(renderer, xOffset0, scanlineY, xOffset1, scan-
lineY);
              xOffset0 += dx0;
              xOffset1 += dx1;
        }
  }
```

3.1.18 Метод void DrawBottomFlatTriangle()

Метод производит отрисовку треугольника, у которого плоская часть снизу. Алгоритм заключается в том, что треугольник заполняется снизу-вверх лииями.
Описание:

```
void DrawBottomFlatTriangle(SDL_Renderer *renderer, SDL_Point *v)
  {
        /*
             0
             /\
            /  \
           /    \
```

```
            /     \
                /       \
             1 ---------- 2
            */
            float dx0 = (float)(v[1].x - v[0].x) / (float)(v[1].y - v[0].y);
            float dx1 = (float)(v[2].x - v[0].x) / (float)(v[2].y - v[0].y);

            float xOffset0 = v[0].x;
            float xOffset1 = v[0].x;

            for (int scanlineY = v[0].y; scanlineY <= v[1].y; scanlineY++)
            {
                SDL_RenderDrawLine(renderer, xOffset0, scanlineY, xOffset1, scan-
lineY);
                xOffset0 += dx0;
                xOffset1 += dx1;
            }
        }
```

3.1.19 Метод void DrawFilledTriangle2D()

Метод производит разделение треугольника на 2 и последующую отрисовку.

Описание:

```
void DrawFilledTriangle2D(SDL_Renderer *renderer, Triangle2D tr) {
    // Points are Integers
    // p[0] need to have lowest y among points
    if (!(tr.p[0].y == tr.p[1].y && tr.p[1].y == tr.p[2].y)) {
        bool sorted = false;
        while (!sorted)
        {
            sorted = true;
            for (int i = 0; i < 2; i++) {
                if (tr.p[i].y > tr.p[i + 1].y)
                {
                    sorted = false;
                    SDL_Point buffPoint = tr.p[i];
                    tr.p[i] = tr.p[i + 1];
                    tr.p[i + 1] = buffPoint;
                }
            }
        }

        if (tr.p[1].y == tr.p[2].y) {
            DrawBottomFlatTriangle(renderer, tr.p);
        } else
            if (tr.p[0].y == tr.p[1].y) {
```

```
                    DrawTopFlatTriangle(renderer, tr.p);
                } else {
                    SDL_Point splitPoint;
                    splitPoint.x = tr.p[0].x + ((float)(tr.p[1].y - tr.p[0].y) / (float)(tr.p[2].y - tr.p[0].y)) * (tr.p[2].x - tr.p[0].x);
                    splitPoint.y = tr.p[1].y;
                    SDL_Point points[3];
                    points[0] = tr.p[0];
                    points[1] = tr.p[1];
                    points[2] = splitPoint;
                    DrawBottomFlatTriangle(renderer, points);
                    points[0] = tr.p[1];
                    points[1] = splitPoint;
                    points[2] = tr.p[2];
                    DrawTopFlatTriangle(renderer, points);
                }
            }

    }
```

3.1.20 Метод void DrawSceneObjects()

Данный метод является основным методом отрисовки. В нем сосредоточены все требующиеся вычисления для корректного рендеринга. Включая клиппинг – обрезание полигонов краями экрана.

Алгоритм:

1) Создание матриц поворота
2) Создание мировой матрицы
3) Создание вектора из полигонов, которые требуется отрисовать
4) Рендеринг

Описание:

```
void DrawSceneObjects(SDL_Renderer *renderer) {
    Matrix4 matRotX, matRotY, matRotZ;

    /*static float theta = 0;
    theta += 1.0f / 30.0f;*/

    // Rotation X
    matRotX = Matrix4_MakeRotationX(0);

    // Rotation Y
    matRotY = Matrix4_MakeRotationY(0);

    // Rotation Z
    matRotZ = Matrix4_MakeRotationZ(0);
```

```
        Matrix4 matTrans;
        matTrans = Matrix4_MakeTranslation(0.0f, 0.0f, 5.0f);

        Matrix4 matWorld;
        matWorld = Matrix4_MakeIdentity();// Form World Matrix
        matWorld = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
        matWorld = Matrix4_MultiplyMatrix(matWorld, matRotZ); // Transform by rotation by Y
        matWorld = Matrix4_MultiplyMatrix(matWorld, matTrans); // Transform by translation

        vec3 upVector = { 0, 1, 0 };
        vec3 targetVector = { 0, 0, 1 };
      Matrix4 m1 = Matrix4_MakeRotationX(MainCamera.fXRotation);
      Matrix4 m2 = Matrix4_MakeRotationY(MainCamera.fYRotation);
        Matrix4 matCameraRot = Matrix4_MultiplyMatrix(m1, m2);
        MainCamera.lookDirection = Matrix4_MultiplyVector(targetVector, matCameraRot);
        targetVector = Vector3_Add(MainCamera.position, MainCamera.lookDirection);
        Matrix4 matCamera = Matrix4_PointAt(MainCamera.position, targetVector, upVector);

        Matrix4 matView = Matrix4_QuickInverse(matCamera);

        // Store triagles for rastering later
        std::vector<Triangle> vecTrianglesToRaster;

        for (GE_Object &obj : GE_DRAW_LIST.obj) {
            //if (!Vector3_Equals(obj.getPosition(), GE_DRAW_LIST.selectorBox.getPosition())) {
                for (Mesh_Side &side : obj.sides) {
                    if (!side.hidden) {
                        for (Triangle &tri : side.mesh.polygons) {
                            FillTrianglesToRasterVector(vecTrianglesToRaster, tri, matWorld, matView);
                        }
                    }
                }
            //}
        }

        for (Mesh_Side &side : GE_DRAW_LIST.selectorBox.sides) {
```

```
                if (!side.hidden) {
                        for (Triangle &tri : side.mesh.polygons) {
                                FillTrianglesToRasterVector(vecTrianglesToRaster,      tri,
    matWorld, matView);
                        }
                }
        }

        sort(vecTrianglesToRaster.begin(),   vecTrianglesToRaster.end(),   [](Triangle
&t1, Triangle &t2)
                {
                        float z1 = (t1.p[0].z + t1.p[1].z + t1.p[2].z) / 3.0f;
                        float z2 = (t2.p[0].z + t2.p[1].z + t2.p[2].z) / 3.0f;
                        return z1 > z2;
                });

        for (Triangle &triToRaster : vecTrianglesToRaster)
        {
                // Clip triangles against all four screen edges, this could yield
                // a bunch of triangles, so create a queue that we traverse to
                //  ensure we only test new triangles generated against planes
                Triangle clipped[2];
                std::list<Triangle> listTriangles;

                // Add initial triangle
                listTriangles.push_back(triToRaster);
                unsigned long nNewTriangles = 1;

                for (int p = 0; p < 4; p++)
                {
                        int nTrisToAdd = 0;
                        while (nNewTriangles > 0)
                        {
                                // Take triangle from front of queue
                                Triangle test = listTriangles.front();
                                listTriangles.pop_front();
                                nNewTriangles--;

                                // Clip it against a plane. We only need to test each
                                // subsequent plane, against subsequent new triangles
                                // as all triangles after a plane clip are guaranteed
                                // to lie on the inside of the plane. I like how this
                                // comment is almost completely and utterly justified
                                switch (p)
```

```
                                {
                                case 0:	nTrisToAdd = Triangle_ClipAgainstPlane({ 0.0f, 0.0f, 0.0f }, { 0.0f, 1.0f, 0.0f }, test, clipped[0], clipped[1]); break;
                                case 1:	nTrisToAdd = Triangle_ClipAgainstPlane({ 0.0f, (float)HEIGHT - 1.0f, 0.0f }, { 0.0f, -1.0f, 0.0f }, test, clipped[0], clipped[1]); break;
                                case 2:	nTrisToAdd = Triangle_ClipAgainstPlane({ 0.0f, 0.0f, 0.0f }, { 1.0f, 0.0f, 0.0f }, test, clipped[0], clipped[1]); break;
                                case 3:	nTrisToAdd = Triangle_ClipAgainstPlane({ (float)WIDTH - 1.0f, 0.0f, 0.0f }, { -1.0f, 0.0f, 0.0f }, test, clipped[0], clipped[1]); break;
                                }

                                // Clipping may yield a variable number of triangles, so
                                // add these new ones to the back of the queue for subsequent
                                // clipping against next planes
                                for (int w = 0; w < nTrisToAdd; w++) {
                                        listTriangles.push_back(clipped[w]);
                                }

                        }
                        nNewTriangles = listTriangles.size();
                }


                // Draw the transformed, viewed, clipped, projected, sorted, clipped triangles
                for (Triangle &t : listTriangles)
                {
                        SDL_Point points[3] = {
                                        { (int)t.p[0].x,(int)t.p[0].y },
                                        { (int)t.p[1].x,(int)t.p[1].y },
                                        { (int)t.p[2].x,(int)t.p[2].y }
                        };
                        Triangle2D tr = { points[0], points[1], points[2] };
                        switch (GE_RENDERING_STYLE)
                        {
                        case Engine3D::RENDERING_STYLES::STD_SHADED:
                                SDL_SetRenderDrawColor(renderer, t.color.R, t.color.G, t.color.B, 255.0f);
                                DrawFilledTriangle2D(renderer, tr);
                                break;
                        case Engine3D::RENDERING_STYLES::STD_POLY_SHADED:
                                SDL_SetRenderDrawColor(renderer, t.color.R, t.color.G, t.color.B, 255.0f);
```

```
                        DrawFilledTriangle2D(renderer, tr);
                        SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
                        DrawTriangle2D(renderer, tr);
                        break;
                    case Engine3D::RENDERING_STYLES::DEBUG_DRAW_ONLY_POLYGONS:
                        SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
                        DrawTriangle2D(renderer, tr);
                        break;
                    default:
                        break;
                    }
                }
            }

        }
```

3.1.21 Метод GE_Object* getGEObjectPointerByPos()

Метод позволяющий получить указатель на объект по его позиции на сцене.

Описание:

```
GE_Object* getGEObjectPointerByPos(vec3 pos) {
    for (GE_Object &obj : GE_DRAW_LIST.obj) {
        if (Vector3_Equals(obj.getPosition(), pos)) {
            return &obj;
        }
    }
    return nullptr;
}
```

3.1.22 Метод void ShowPreviouslyUnneededSidesByObj()

Метод, оптимизирующий рендеринг. Благодаря указанию направления полигона открываются новые полигоны на указанной точке.

Алгоритм:

Проверяется каждая соседняя точка около указанной. Если в ней есть объект, открывается грань в зависимости от направления.

Описание:

```
void ShowPreviouslyUnneededSidesByObj(GE_Object *obj) {
    struct vec3_wd {
        GE_MESH_SIDE_TYPE sideType = GE_MESH_SIDE_TYPE::UNDEFINED;
        GE_MESH_SIDE_TYPE counterSideType = GE_MESH_SIDE_TYPE::UNDEFINED;
        vec3 vec;
    };
    switch (obj->getObjType()) {
    case GE_OBJECT_TYPE::CUBE: {
```

```
                vec3 objPos = obj->getPosition();

                std::vector<vec3_wd> neighbourPositions;
                neighbourPositions.push_back({       GE_MESH_SIDE_TYPE::EAST,
      GE_MESH_SIDE_TYPE::WEST,      1,  0,  0 });
                neighbourPositions.push_back({         GE_MESH_SIDE_TYPE::TOP,
GE_MESH_SIDE_TYPE::BOTTOM,         0,  1,  0 });
                neighbourPositions.push_back({    GE_MESH_SIDE_TYPE::NORTH,
GE_MESH_SIDE_TYPE::SOUTH,    0,  0,  1 });
                neighbourPositions.push_back({       GE_MESH_SIDE_TYPE::WEST,
GE_MESH_SIDE_TYPE::EAST,      -1,  0,  0 });
                neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::BOTTOM,
GE_MESH_SIDE_TYPE::TOP,         0, -1,  0 });
                neighbourPositions.push_back({
GE_MESH_SIDE_TYPE::SOUTH,GE_MESH_SIDE_TYPE::NORTH,      0,  0, -1 });
                for (vec3_wd &v_wd : neighbourPositions) {
                        v_wd.vec = Vector3_Add(v_wd.vec, objPos);
                        GE_Object       *neighbourObj       =       getGEObjectPointerBy-
Pos(v_wd.vec);
                        if (neighbourObj != nullptr) {
                                switch (neighbourObj->getObjType()) {
                                case GE_OBJECT_TYPE::CUBE: {
                                        neighbourObj->showSide(v_wd.counterSideType);
                                        break;
                                }
                                default:
                                        break;
                                }
                        }
                }

                        break;
                }
                default:
                        break;
                }
        }
```

3.1.23 Метод void HideUnneededSidesByObj()

Метод, оптимизирующий рендеринг. Благодаря указанию направления полигона скрываются новые полигоны на указанной точке.

Алгоритм:

Проверяется каждая соседняя точка около указанной. Если в ней есть объект, скрывается грань в зависимости от направления.

Описание:

```
void HideUnneededSidesByObj(GE_Object *obj) {
        struct vec3_wd {
                GE_MESH_SIDE_TYPE sideType = GE_MESH_SIDE_TYPE::UNDEFINED;
                GE_MESH_SIDE_TYPE counterSideType = GE_MESH_SIDE_TYPE::UNDEFINED;
                vec3 vec;
        };
        switch (obj->getObjType()) {
        case GE_OBJECT_TYPE::CUBE: {
                vec3 objPos = obj->getPosition();

                std::vector<vec3_wd> neighbourPositions;
                neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::EAST, GE_MESH_SIDE_TYPE::WEST, 1, 0, 0 });
                neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::TOP, GE_MESH_SIDE_TYPE::BOTTOM, 0, 1, 0 });
                neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::NORTH, GE_MESH_SIDE_TYPE::SOUTH, 0, 0, 1 });
                neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::WEST, GE_MESH_SIDE_TYPE::EAST, -1, 0, 0 });
                neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::BOTTOM, GE_MESH_SIDE_TYPE::TOP, 0, -1, 0 });
                neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::SOUTH,GE_MESH_SIDE_TYPE::NORTH, 0, 0, -1 });
                for (vec3_wd &v_wd : neighbourPositions) {
                        v_wd.vec = Vector3_Add(v_wd.vec, objPos);
                        GE_Object *neighbourObj = getGEObjectPointerByPos(v_wd.vec);
                        if (neighbourObj != nullptr) {
                                switch (neighbourObj->getObjType()) {
                                case GE_OBJECT_TYPE::CUBE: {
                                        neighbourObj->hideSide(v_wd.counterSideType);
                                        obj->hideSide(v_wd.sideType);
                                        break;
                                }
```

```
                        default:
                            break;
                        }
                    }
                }

                break;
            }
            default:
                break;
            }
        }
```

3.1.24 Метод void CreateBlockAtSelectorPosition()

Метод, указывающий движку создать новый объект на позиции селектора.

Описание:

```
    void CreateBlockAtSelectorPosition() {
        vec3 pos = GE_DRAW_LIST.selectorBox.getPosition();
        GE_Object *obj = getGEObjectPointerByPos(pos);
        if (obj == nullptr){
            GE_Object sBox = GE_STD_OBJECTS.CUBE;
            sBox.moveTo(pos);
            HideUnneededSidesByObj(&sBox);
            GE_DRAW_LIST.obj.push_back(sBox);
            printf("Created block at %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
        } else {
            printf("Aborted to created block at %.2f %.2f %.2f\n", pos.x, pos.y,
pos.z);
        }
    }
```

3.1.25 Метод void RemoveBlockAtSelectorPosition()

Метод, указывающий движку удалить объект на позиции селектора.

Описание:

```
    void RemoveBlockAtSelectorPosition() {
        // Can't use getGEObjectPointerByPos() because we need to remove from vec-
tor by index
        vec3 pos = GE_DRAW_LIST.selectorBox.getPosition();
        int index = -1;
        int i = 0;
        for (GE_Object &obj : GE_DRAW_LIST.obj) {
            if (Vector3_Equals(obj.getPosition(), pos)) {
                index = i;
                break;
            }
            i++;
```

```
        }
        if (index >= 0)      {
            ShowPreviouslyUnneededSidesByObj(&(GE_DRAW_LIST.obj[in-
dex]));
            GE_DRAW_LIST.obj.erase(GE_DRAW_LIST.obj.begin() + index);
            printf("Removed block at %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
        } else {
            printf("Unable to remove block at %.2f %.2f %.2f\n", pos.x, pos.y,
pos.z);
        }
    }
```

3.1.26 Метод void ChangeBlockColorAtSelectorPosition()

Метод, указывающий движку поменять цвет объекта на позиции селектора.

Описание:

```
void ChangeBlockColorAtSelectorPosition() {
        vec3 pos = GE_DRAW_LIST.selectorBox.getPosition();
        GE_Object *obj = getGEObjectPointerByPos(pos);
        if (obj != nullptr) {
            Colors::Types newColorType = GE_COLORS.getColorTypeAfter(obj-
>getColorType());
            GE_Color newColor = GE_COLORS.getColorByType(newColor-
Type);
            obj->setColor(newColor, newColorType);
            printf("Changed block color at %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
        }
        else {
            printf("Aborted to change block color at %.2f %.2f %.2f\n", pos.x, pos.y,
pos.z);
        }
    }
```

3.1.27 Метод void StartRenderLoop()

Метод, который создает рендерер – обьект, использующийся для рендеринге на экране. Также здесь обновляются характеристики окна.

Описание:

```
void StartRenderLoop() {
        SDL_Renderer *renderer = SDL_CreateRenderer(window, -1, 0);
        Uint32 start;
        SDL_Event windowEvent;
        while (isRunning)
        {
            if (SDL_PollEvent(&windowEvent))
            {
                if (windowEvent.type == SDL_QUIT)
                {
```

```
                    isRunning = false;
                }

                if (windowEvent.type == SDL_KEYDOWN) {
                    switch (GE_CURRENT_KEYBOARD_CONTROL) {
                        case Engine3D::KEYBOARD_CONTROL_TYPES::ALLOW_SCENE_EDITING:
                            SceneEditingHandle(windowEvent.key.keysym.scancode);
                            break;
                        case Engine3D::KEYBOARD_CONTROL_TYPES::ALLOW_OBJECT_EDITING:
                            break;
                        case Engine3D::KEYBOARD_CONTROL_TYPES::ALLOW_CAMERA_CONTROL:
                            CameraMovementHandle(windowEvent.key.keysym.scancode);
                            break;
                        default:
                            break;
                        }
                    }
                }
                start = SDL_GetTicks();

                //Updates properties of the screen and camera
                updateScreenAndCameraProperties(renderer);

                //Background(Clears with color)
                SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
                SDL_RenderClear(renderer);

                //Draws scene
                SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
                DrawSceneObjects(renderer);

                // Renders window
                SDL_RenderPresent(renderer);
                Uint32 ticks = SDL_GetTicks();
                if (1000 / FRAMES_PER_SECOND > ticks - start) {
                    SDL_Delay(1000 / FRAMES_PER_SECOND - (ticks - start));
                }
            }
            SDL_DestroyRenderer(renderer);
```

}

3.1.28 Метод void CreateCubicFormByTopMesh()

Метод, позволяющий создать полигональную сетку кубической формы путем указания только верхней полигональной составляющей фигуры.

Алгоритм:

1) Создается верхняя полигональная сетка
2) С помощью матриц поворота формируются сетки для каждой грани

Описание:

```
void CreateCubicFormByTopMesh(GE_Object &buffObj, Mesh &Mesh_TOP) {
	Mesh_Side buffSide;

	Matrix4 matRotX;
	Matrix4 matRotY;
	Matrix4 matRotZ;
	Matrix4 matRot;

	buffSide.type = GE_MESH_SIDE_TYPE::TOP;
	buffSide.mesh = Mesh_TOP;
	matRotX = Matrix4_MakeRotationX(0);
	matRotY = Matrix4_MakeRotationY(0);
	matRotZ = Matrix4_MakeRotationZ(0);
	matRot = Matrix4_MakeIdentity();
	matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
	matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
	for (Triangle &polygon : buffSide.mesh.polygons) {
		for (vec3 &v : polygon.p) {
			v = Matrix4_MultiplyVector(v, matRot);
		}
	}
	buffObj.sides.push_back(buffSide);


	buffSide.type = GE_MESH_SIDE_TYPE::BOTTOM;
	buffSide.mesh = Mesh_TOP;
	matRotX = Matrix4_MakeRotationX(0);
	matRotY = Matrix4_MakeRotationY(0);
	matRotZ = Matrix4_MakeRotationZ(M_PI);
	matRot = Matrix4_MakeIdentity();
	matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
	matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
```

```
		for (Triangle &polygon : buffSide.mesh.polygons) {
			for (vec3 &v : polygon.p) {
				v = Matrix4_MultiplyVector(v, matRot);
			}
		}
		buffObj.sides.push_back(buffSide);

		buffSide.type = GE_MESH_SIDE_TYPE::WEST;
		buffSide.mesh = Mesh_TOP;
		matRotX = Matrix4_MakeRotationX(0);
		matRotY = Matrix4_MakeRotationY(0);
		matRotZ = Matrix4_MakeRotationZ(M_PI / 2.0f);
		matRot = Matrix4_MakeIdentity();
		matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
		matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
		for (Triangle &polygon : buffSide.mesh.polygons) {
			for (vec3 &v : polygon.p) {
				v = Matrix4_MultiplyVector(v, matRot);
			}
		}
		buffObj.sides.push_back(buffSide);

		buffSide.type = GE_MESH_SIDE_TYPE::EAST;
		buffSide.mesh = Mesh_TOP;
		matRotX = Matrix4_MakeRotationX(0);
		matRotY = Matrix4_MakeRotationY(0);
		matRotZ = Matrix4_MakeRotationZ(-M_PI / 2.0f);
		matRot = Matrix4_MakeIdentity();
		matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
		matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
		for (Triangle &polygon : buffSide.mesh.polygons) {
			for (vec3 &v : polygon.p) {
				v = Matrix4_MultiplyVector(v, matRot);
			}
		}
		buffObj.sides.push_back(buffSide);

		buffSide.type = GE_MESH_SIDE_TYPE::NORTH;
		buffSide.mesh = Mesh_TOP;
		matRotX = Matrix4_MakeRotationX(M_PI / 2.0f);
```

```
            matRotY = Matrix4_MakeRotationY(0);
            matRotZ = Matrix4_MakeRotationZ(0);
            matRot = Matrix4_MakeIdentity();
            matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
            matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
            for (Triangle &polygon : buffSide.mesh.polygons) {
                for (vec3 &v : polygon.p) {
                    v = Matrix4_MultiplyVector(v, matRot);
                }
            }
            buffObj.sides.push_back(buffSide);

            buffSide.type = GE_MESH_SIDE_TYPE::SOUTH;
            buffSide.mesh = Mesh_TOP;
            matRotX = Matrix4_MakeRotationX(-M_PI / 2.0f);
            matRotY = Matrix4_MakeRotationY(0);
            matRotZ = Matrix4_MakeRotationZ(0);
            matRot = Matrix4_MakeIdentity();
            matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
            matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
            for (Triangle &polygon : buffSide.mesh.polygons) {
                for (vec3 &v : polygon.p) {
                    v = Matrix4_MultiplyVector(v, matRot);
                }
            }
            buffObj.sides.push_back(buffSide);
    }
```

3.1.29. Метод void startScene()

Метод, указывающий движку о том, что требуется начать рендеринг.

Описание:

```
void startScene() {
        if (GE_ERROR_CODE != ERROR_CODES::ZERO) {
            printf("Unable to start scene! ERROR: %d\n", GE_ERROR_CODE);
            return;
        }
        resetMainCamera();
        printf("Reset camera!\n");
        isRunning = true;
        StartRenderLoop();}
```

4 ТЕСТИРОВАНИЕ, ПРОВЕРКА РАБОТОСПОСОБНОСТИ И АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

Для того, чтобы соответствовать требованиям к проектируемому программному средству, необходимо, чтобы оно прошло некоторое тестирование, способное выявить его недостатки.

Таблица 4.1 – Тестирование программы

№ Теста	Тестируемая функциональность	Последовательность действий	Ожидаемый результат	Полученный результат
1	Создание объекта	Нажатие кнопки F	Создан новый куб	Тест пройден
2	Удаление объекта	Нажатие кнопки R	Удален куб	Тест пройден
3	Смена цвета объекта	Нажатие кнопки E	Куб сменил цвет	Тест пройден
4	Перемещение объекта	Нажатие кнопки “стрелка” влево	Куб переместился	Тест пройден
5	Перемещение объекта	Нажатие кнопки “стрелка” вправо	Куб переместился	Тест пройден
6	Перемещение объекта	Нажатие кнопки “стрелка” вверх	Куб переместился	Тест пройден
7	Перемещение объекта	Нажатие кнопки “стрелка” вниз	Куб переместился	Тест пройден
8	Переход в другой режим управления	Нажатие кнопки ~	Появилась возможность управлять камерой	Тест пройден
9	Перемещение камеры	Нажатие кнопки W	Камера переместилась	Тест пройден
10	Перемещение камеры	Нажатие кнопки A	Камера переместилась	Тест пройден
11	Перемещение камеры	Нажатие кнопки S	Камера переместилась	Тест пройден
12	Перемещение камеры	Нажатие кнопки D	Камера переместилась	Тест пройден

13	Поворот камеры	Нажатие кнопки “стрелка” влево	Камера совершила поворот	Тест пройден
14	Поворот камеры	Нажатие кнопки “стрелка” вправо	Камера совершила поворот	Тест пройден
15	Поворот камеры	Нажатие кнопки “стрелка” вверх	Камера совершила поворот	Тест пройден
16	Поворот камеры	Нажатие кнопки “стрелка” вниз	Камера совершила поворот	Тест пройден
17	Изменение FOV	Нажатие кнопки [	Изменился угол обзора	Тест пройден
18	Изменение FOV	Нажатие кнопки]	Изменился угол обзора	Тест пройден

5 РУКОВОДСТВО ПО УСТАНОВКЕ И ИСПОЛЬЗОВАНИЮ

5.1 Основные требования для запуска данного программного средства:

- ОС: Версия Microsoft Windows от XP и выше, Mac OS Lion и выше;
- Процессор: Pentium® III 800 МГц или AMD Athlon;
- RAM: От 1 Гб;
- Место на диске: от 3 Мб свободного места.

Исходя из данных требований следует, что данная программа может запускаться практически на любом компьютере.

5.2 Руководство по установке

Установка программы не требуется, т.к она является переносимым приложением.

Переносимое приложение — программное обеспечение, которое для своего запуска не требует процедуры установки и может полностью храниться на съёмных носителях информации, что позволяет использовать данное ПО на многих компьютерах.

5.3 Руководство по использованию

1) Запустить программу.

Рисунок 5.1 – Скриншот запущенной программы

2) Построить желаемую модель.

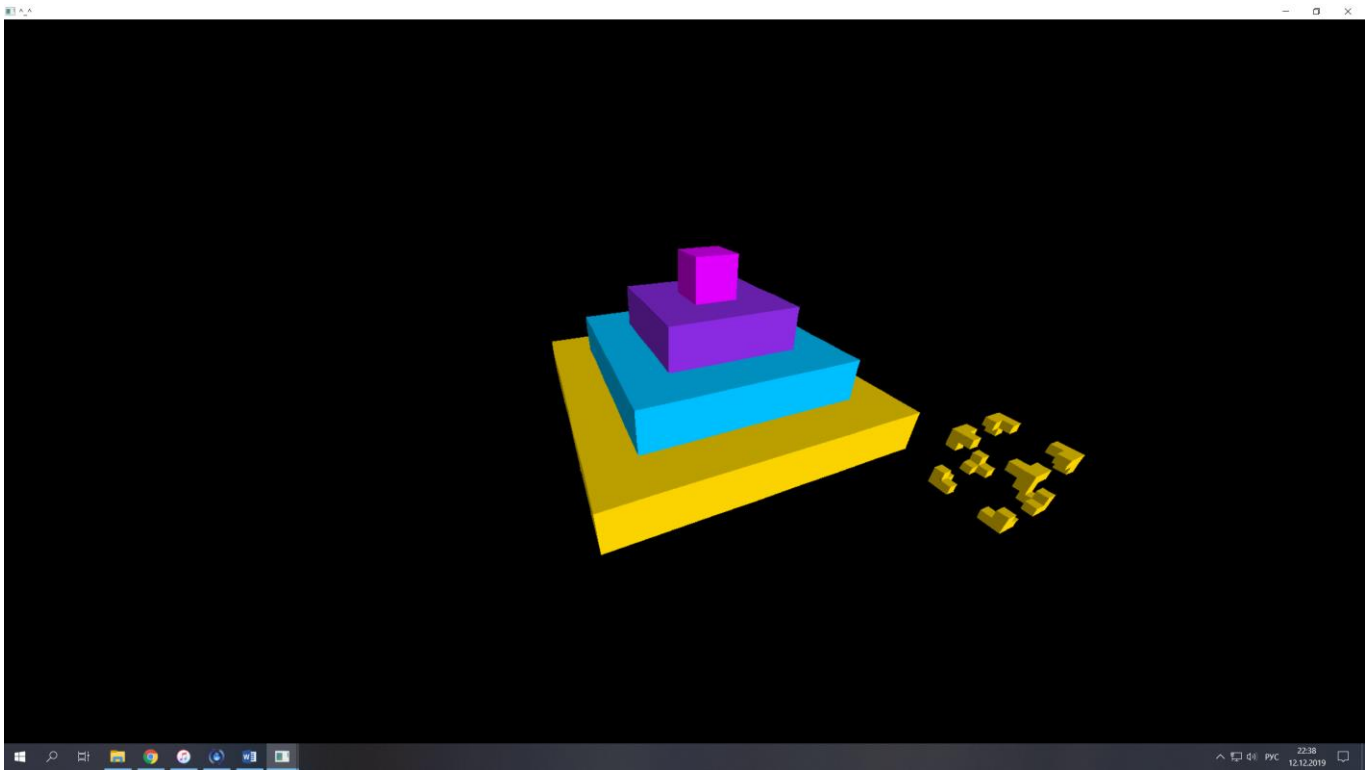


Рисунок 5.2 – Скриншот с построенной фигурой

3) Закрыть программу.

ЗАКЛЮЧЕНИЕ

В настоящее время вычислительные мощности компьютеров растут невероятно быстро. Благодаря столь быстрому росту компьютеры обрели возможность выйти за рамки 2D пространства. Сегодня самый простой персональный компьютер способен на несложные 3D вычисления. Мощные же компьютеры способны проводить космические симуляции с высокой точность.

Благодаря 3D графическим редакторам у людей появилась возможность создавать трехмерные объекты, которые могут быть использованы в кино и играх, а также могут быть перенесены в реальный мир с помощью 3D принтера.

В рамках данного курсового проекта было разработано программное средство, позволяющее создавать несложные геометрические фигуры, которые в последующем могут быть экспортированы и использованы в других программах.

Для воплощения проекта в жизнь потребовалось изучить основы компьютерной графики и векторную алгебру.

Существует много возможностей для дальнейшего развития приложения. Некоторыми из них являются добавление новых фигур, возможность создания произвольной фигуры внутри приложения, добавление поддержки популярных форматов 3D-моделей. Также существует возможность добавить функционал игрового движка с помощью создания физического движка и внедрения его в написанный редактор.

СПИСОК ЛИТЕРАТУРЫ

[1] microsoft.com [Электронный портал]. – Электронные данные. – Режим доступа: https://microsoft.com/ru/

[2] blender.com [Электронный портал]. – Электронные данные. – Режим доступа: https:/blender.com/start.html

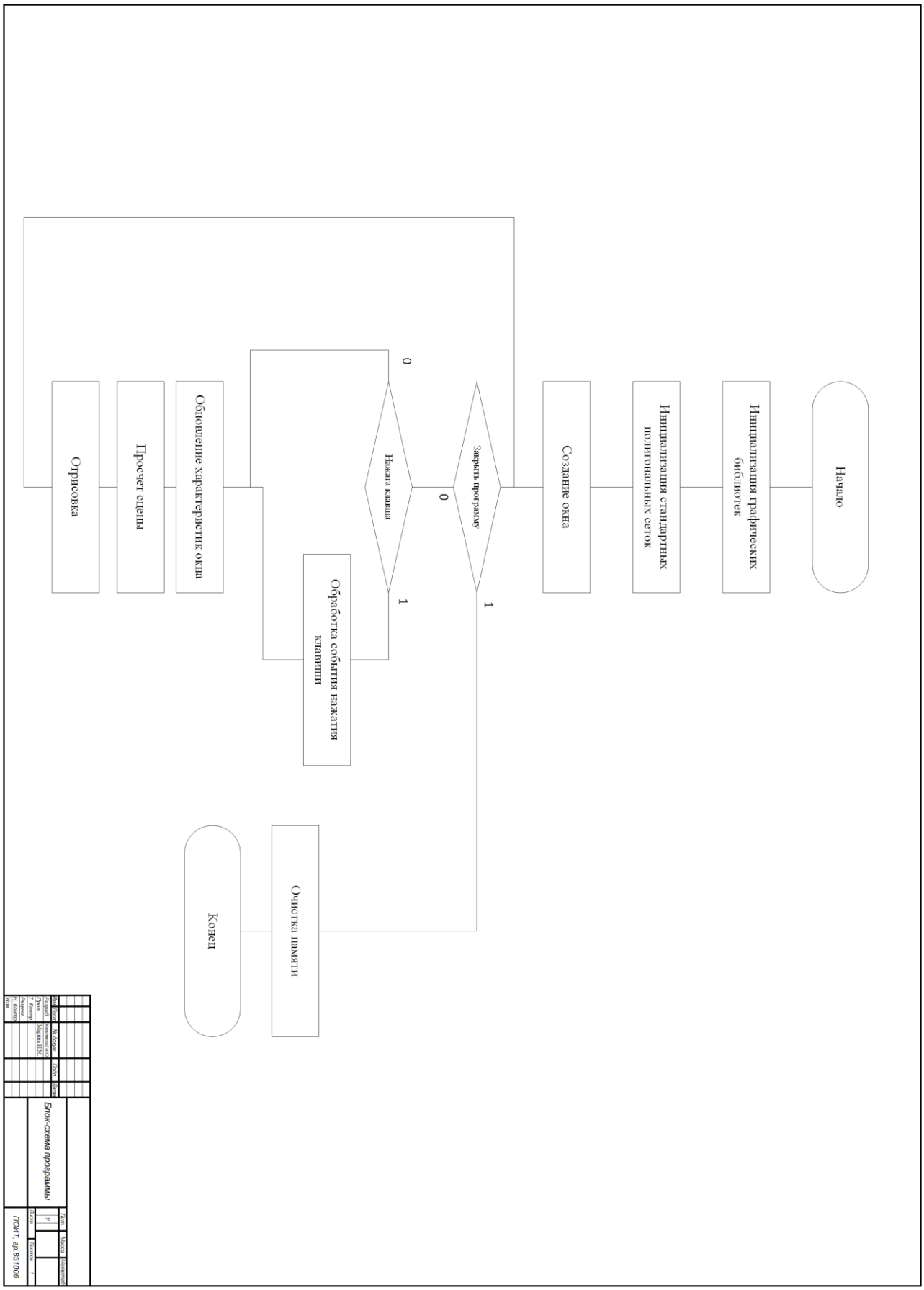
[3] autodesk.com [Электронный портал]. – Электронные данные. – Режим доступа: https://autodesk.com/3dsmax/

[4] matesha.ru [Электронный портал]. – Электронные данные. – Режим доступа: https://matesha.ru/vector/

[5] chikibamboni.net [Электронный портал]. – Электронные данные. – Режим доступа: https://chikibamboni.net/graphics/

ПРИЛОЖЕНИЯ

Приложение 1 (схема алгоритма программы на А1)



Приложение 2 (Код программы)

```
//
//  main.cpp
//  3DGE _SDL2
//
//  Created by Михаил Ковалевский on 19.10.2019.
//  Copyright © 2019 Mikhail Kavaleuski. All rights reserved.
//

#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
#include "GE_3DMath.h"
#ifdef _WIN32
	//define something for Windows (32-bit and 64-bit, this part is common)
	#include <SDL.h>
#elif __APPLE__
	//define something for MacOS
	#include <SDL2/SDL.h>
#endif




class Engine3D {
private:
	enum class ERROR_CODES {
		ZERO, // No errors
		WINDOW_INIT_ERROR, // Error while window initialization
		SDL2_INIT_ERROR // Error while SDL2 initialization
	};
	ERROR_CODES GE_ERROR_CODE = ERROR_CODES::ZERO;

	enum class KEYBOARD_CONTROL_TYPES {
		ALLOW_SCENE_EDITING,
		ALLOW_OBJECT_EDITING,
		ALLOW_CAMERA_CONTROL
	};
	KEYBOARD_CONTROL_TYPES GE_CURRENT_KEYBOARD_CONTROL = KEYBOARD_CON-
TROL_TYPES::ALLOW_SCENE_EDITING;

	enum class RENDERING_STYLES {
		STD_SHADED,
		STD_POLY_SHADED,
		DEBUG_DRAW_ONLY_POLYGONS
	};
	RENDERING_STYLES GE_RENDERING_STYLE = RENDERING_STYLES::STD_SHADED;

	enum class GE_OBJECT_TYPE {
		UNDEFINED,
		SELECTOR,
		CUBE
	};

	enum class GE_MESH_SIDE_TYPE {
		UNDEFINED,
		SOUTH,
		EAST,
		NORTH,
```

```
	WEST,
	TOP,
	BOTTOM
};

struct GE_Color {
	float R;
	float G;
	float B;
};

struct Colors {
private:
	GE_Color WHITE = { 255.0f, 255.0f, 255.0f };
	GE_Color YELLOW = { 250.0f, 211.0f, 0.0f };
	GE_Color BLUE = { 32.0f, 5.0f, 137.0f };
	GE_Color AQUA = { 0.0f, 191.0f, 255.0f };
	GE_Color VIOLET = { 138.0f, 43.0f, 226.0f };
	GE_Color PURPLE = { 223.0f, 0.0f, 254.0f };
	GE_Color GREEN = { 62.0f, 255.0f, 21.0f };
	GE_Color RED = { 254.0f, 0.0f, 0.0f };
public:
	enum class Types {
		WHITE,
		YELLOW,
		BLUE,
		AQUA,
		VIOLET,
		PURPLE,
		GREEN,
		RED
	};
	GE_Color getColorByType(Types t) {
		// Invalid type will return WHITE
		switch (t) {
			case Types::WHITE: return WHITE; break;
			case Types::YELLOW: return YELLOW; break;
			case Types::BLUE: return BLUE; break;
			case Types::AQUA: return AQUA; break;
			case Types::VIOLET: return VIOLET; break;
			case Types::PURPLE: return PURPLE; break;
			case Types::GREEN: return GREEN; break;
			case Types::RED: return RED; break;
			default: return WHITE; break;
		}
	}
	Types getColorTypeAfter(Types t) {
		// Invalid type will return WHITE
		switch (t) {
			case Types::WHITE: return Types::YELLOW; break;
			case Types::YELLOW: return Types::BLUE; break;
			case Types::BLUE: return Types::AQUA; break;
			case Types::AQUA: return Types::VIOLET; break;
			case Types::VIOLET: return Types::PURPLE; break;
			case Types::PURPLE: return Types::GREEN; break;
			case Types::GREEN: return Types::RED; break;
			case Types::RED: return Types::WHITE; break;
			default: return Types::WHITE; break;
		}
	}
};
Colors GE_COLORS;
```

```
struct Triangle {
	vec3 p[3];
	GE_Color color = { 255.0f, 255.0f, 255.0f };
};

struct Triangle2D {
	SDL_Point p[3];
};

struct Mesh {
	std::vector<Triangle> polygons; // First is always considered as a selector
};

struct Mesh_Side {
	GE_MESH_SIDE_TYPE type = GE_MESH_SIDE_TYPE::UNDEFINED;
	bool hidden = false;
	Mesh mesh;
};

struct GE_Object {
private:
	vec3 position = { 0, 0, 0 };
	GE_OBJECT_TYPE objType = GE_OBJECT_TYPE::UNDEFINED;
	Colors::Types colorType = Colors::Types::WHITE;
public:
	std::vector<Mesh_Side> sides;

	vec3 getPosition() {
		return position;
	}

	void setObjType(GE_OBJECT_TYPE t) {
		objType = t;
	}

	GE_OBJECT_TYPE getObjType() {
		return objType;
	}

	Colors::Types getColorType() {
		return colorType;
	}

	void moveBy(vec3 v) {
		position = Vector3_Add(position, v);
		for (Mesh_Side &side : sides) {
			for (Triangle &tri : side.mesh.polygons) {
				for (vec3 &p : tri.p) {
					p = Vector3_Add(p, v);
				}
			}
		}
		printf("Moved block to %.2f %.2f %.2f\n", position.x, position.y, position.z);
	}

	void moveTo(vec3 v) {
		vec3 offset = Vector3_Sub(v, position);
		moveBy(offset);
	}

	void scaleBy(float k) {
```

```
		for (Mesh_Side &side : sides) {
			for (Triangle &tri : side.mesh.polygons) {
				for (vec3 &p : tri.p) {
					p = Vector3_Mul(p, k);
				}
			}
		}
		printf("Scaled block by %.2f\n", k);
	}

	void setColor(GE_Color _color, Colors::Types _t) {
		colorType = _t;
		for (Mesh_Side &side : sides) {
			for (Triangle &tri : side.mesh.polygons) {
				tri.color = _color;
			}
		}
		printf("Changed color to %.2f %.2f %.2f\n", _color.R, _color.G, _color.B);
	}

	void hideSide(GE_MESH_SIDE_TYPE type) {
		for (Mesh_Side &side : sides) {
			if (side.type == type) {
				side.hidden = true;
				break;
			}
		}
	}

	void showSide(GE_MESH_SIDE_TYPE type) {
		for (Mesh_Side &side : sides) {
			if (side.type == type) {
				side.hidden = false;
				break;
			}
		}
	}
};

struct DrawList {
	GE_Object selectorBox;
	std::vector<GE_Object> obj;
};
DrawList GE_DRAW_LIST;

struct GE_Camera {
	// pi -3.14159f
	vec3 position = { 0.0f, 0.0f, 0.0f };
	vec3 lookDirection = { 0.0f, 0.0f, 1.0f };
	float fXRotation = 0.0f;
	float fYRotation = 0.0f;
	float fFOV = 90.0f;
	float fNear = 0.1f;
	float fFar = 1000.0f;
};

struct GE_STD_OBJECT_TYPES {
	GE_Object SELECTOR;
	GE_Object CUBE;
};
GE_STD_OBJECT_TYPES GE_STD_OBJECTS;
```

```
	Matrix4 matProj;
	GE_Camera MainCamera;

	// Illumination
	vec3 LightDirection = { 0.5f, 0.75f, -1.0f }; // LIGHT ORIGIN
	vec3 ObjectMeshAnchor = { -0.5, -0.5, -0.5 }; // Required for objects to have (0,0,0) in middle of them

	const char title[4] = "^_^";
	const int FRAMES_PER_SECOND = 120;

	int WIDTH, HEIGHT;

	bool isRunning = false;

	SDL_Window *window = NULL;

	bool check_window(SDL_Window *window) {
		if (window == NULL)
		{
			printf("Error creating window! SDL_Error: %s\n", SDL_GetError());
			return false;
		} else {
			printf("Created window(%dx%d)!\n", WIDTH, HEIGHT);
			return true;
		}
	}

	int Triangle_ClipAgainstPlane(vec3 plane_p, vec3 plane_n, Triangle &in_tri, Triangle &out_tri1, Triangle &out_tri2)
	{
		// Make sure plane normal is indeed normal
		plane_n = Vector3_Normalize(plane_n);

		// Return signed shortest distance from point to plane, plane normal must be normalised
		auto dist = [&](vec3 &p)
		{
			//vec3 n = Vector3_Normalize(p);
			return (plane_n.x * p.x + plane_n.y * p.y + plane_n.z * p.z - Vector3_DotProduct(plane_n,
plane_p));
		};

		// Create two temporary storage arrays to classify points either side of plane
		// If distance sign is positive, point lies on "inside" of plane
		vec3 *inside_points[3];  int nInsidePointCount = 0;
		vec3 *outside_points[3]; int nOutsidePointCount = 0;

		// Get signed distance of each point in triangle to plane
		float d0 = dist(in_tri.p[0]);
		float d1 = dist(in_tri.p[1]);
		float d2 = dist(in_tri.p[2]);

		if (d0 >= 0) { inside_points[nInsidePointCount++] = &in_tri.p[0]; }
		else { outside_points[nOutsidePointCount++] = &in_tri.p[0]; }
		if (d1 >= 0) { inside_points[nInsidePointCount++] = &in_tri.p[1]; }
		else { outside_points[nOutsidePointCount++] = &in_tri.p[1]; }
		if (d2 >= 0) { inside_points[nInsidePointCount++] = &in_tri.p[2]; }
		else { outside_points[nOutsidePointCount++] = &in_tri.p[2]; }

		// Now classify triangle points, and break the input triangle into
		// smaller output triangles if required. There are four possible
		// outcomes...

		if (nInsidePointCount == 0)
```

```
{
	// All points lie on the outside of plane, so clip whole triangle
	// It ceases to exist

	return 0; // No returned triangles are valid
}

if (nInsidePointCount == 3)
{
	// All points lie on the inside of plane, so do nothing
	// and allow the triangle to simply pass through
	out_tri1 = in_tri;

	return 1; // Just the one returned original triangle is valid
}

if (nInsidePointCount == 1 && nOutsidePointCount == 2)
{
	// Triangle should be clipped. As two points lie outside
	// the plane, the triangle simply becomes a smaller triangle

	// Copy appearance info to new triangle
	/*out_tri1.col = in_tri.col;
	out_tri1.sym = in_tri.sym;*/
	/*out_tri1.R = in_tri.R;
	out_tri1.G = in_tri.G;
	out_tri1.B = in_tri.B;*/
	out_tri1.color = in_tri.color;

	// The inside point is valid, so keep that...
	out_tri1.p[0] = *inside_points[0];

	// but the two new points are at the locations where the
	// original sides of the triangle (lines) intersect with the plane
	out_tri1.p[1] = Vector3_IntersectPlane(plane_p, plane_n, *inside_points[0], *outside_points[0]);
	out_tri1.p[2] = Vector3_IntersectPlane(plane_p, plane_n, *inside_points[0], *outside_points[1]);

	return 1; // Return the newly formed single triangle
}

if (nInsidePointCount == 2 && nOutsidePointCount == 1)
{
	// Triangle should be clipped. As two points lie inside the plane,
	// the clipped triangle becomes a "quad". Fortunately, we can
	// represent a quad with two new triangles

	// Copy appearance info to new triangles
	/*out_tri1.col = in_tri.col;
	out_tri1.sym = in_tri.sym;

	out_tri2.col = in_tri.col;
	out_tri2.sym = in_tri.sym;*/
	/*out_tri1.R = in_tri.R;
	out_tri1.G = in_tri.G;
	out_tri1.B = in_tri.B;*/
	out_tri1.color = in_tri.color;

	/*out_tri2.R = in_tri.R;
	out_tri2.G = in_tri.G;
	out_tri2.B = in_tri.B;*/
	out_tri2.color = in_tri.color;
```

```
                        // The first triangle consists of the two inside points and a new
                        // point determined by the location where one side of the triangle
                        // intersects with the plane
                        out_tri1.p[0] = *inside_points[0];
                        out_tri1.p[1] = *inside_points[1];
                        out_tri1.p[2] = Vector3_IntersectPlane(plane_p, plane_n, *inside_points[0], *outside_points[0]);

                        // The second triangle is composed of one of he inside points, a
                        // new point determined by the intersection of the other side of the
                        // triangle and the plane, and the newly created point above
                        out_tri2.p[0] = *inside_points[1];
                        out_tri2.p[1] = out_tri1.p[2];
                        out_tri2.p[2] = Vector3_IntersectPlane(plane_p, plane_n, *inside_points[1], *outside_points[0]);

                        return 2; // Return two newly formed triangles which form a quad
                }

    return -1;
        }

        void updateScreenAndCameraProperties(SDL_Renderer *renderer) {
                // Gets real size of the window(Fix for MacOS/Resizing)
                SDL_GetRendererOutputSize(renderer, &WIDTH, &HEIGHT);
                const float fAspectRatio = (float)HEIGHT / (float)WIDTH;
                matProj = Matrix4_MakeProjection(MainCamera.fFOV, fAspectRatio, MainCamera.fNear, Main-
Camera.fFar);
        }

        void DrawTriangle2D(SDL_Renderer *renderer, Triangle2D tr) {
                SDL_Point points[4] = { tr.p[0], tr.p[1], tr.p[2], tr.p[0] };
                SDL_RenderDrawLines(renderer, points, 4);
        }

        void DrawTopFlatTriangle(SDL_Renderer *renderer, SDL_Point *v)
        {
                /*
                  0 ---------- 1
                          \              /
                           \     /
                            \           /
                             \ /
                              \/
                                      2
                */
                float dx0 = (float)(v[0].x - v[2].x) / (float)(v[2].y - v[0].y);
                float dx1 = (float)(v[1].x - v[2].x) / (float)(v[2].y - v[1].y);

                float xOffset0 = v[2].x;
                float xOffset1 = v[2].x;

                for (int scanlineY = v[2].y; scanlineY >= v[0].y; scanlineY--)
                {
                        SDL_RenderDrawLine(renderer, xOffset0, scanlineY, xOffset1, scanlineY);
                        xOffset0 += dx0;
                        xOffset1 += dx1;
                }
        }

        void DrawBottomFlatTriangle(SDL_Renderer *renderer, SDL_Point *v)
        {
                /*
                     0
```

```
                    /\
                   /  \
                  /    \
                 /      \
                /        \
               1 ---------- 2
              */
              float dx0 = (float)(v[1].x - v[0].x) / (float)(v[1].y - v[0].y);
              float dx1 = (float)(v[2].x - v[0].x) / (float)(v[2].y - v[0].y);

              float xOffset0 = v[0].x;
              float xOffset1 = v[0].x;

              for (int scanlineY = v[0].y; scanlineY <= v[1].y; scanlineY++)
              {
                    SDL_RenderDrawLine(renderer, xOffset0, scanlineY, xOffset1, scanlineY);
                    xOffset0 += dx0;
                    xOffset1 += dx1;
              }
        }

        void DrawFilledTriangle2D(SDL_Renderer *renderer, Triangle2D tr) {
              // Points are Integers
              // p[0] need to have lowest y among points
              if (!(tr.p[0].y == tr.p[1].y && tr.p[1].y == tr.p[2].y)) {
                    bool sorted = false;
                    while (!sorted)
                    {
                          sorted = true;
                          for (int i = 0; i < 2; i++) {
                                if (tr.p[i].y > tr.p[i + 1].y)
                                {
                                      sorted = false;
                                      SDL_Point buffPoint = tr.p[i];
                                      tr.p[i] = tr.p[i + 1];
                                      tr.p[i + 1] = buffPoint;
                                }
                          }
                    }

                    if (tr.p[1].y == tr.p[2].y) {
                          DrawBottomFlatTriangle(renderer, tr.p);
                    } else
                          if (tr.p[0].y == tr.p[1].y) {
                                DrawTopFlatTriangle(renderer, tr.p);
                          } else {
                                SDL_Point splitPoint;
                                splitPoint.x = tr.p[0].x + ((float)(tr.p[1].y - tr.p[0].y) / (float)(tr.p[2].y - tr.p[0].y))
* (tr.p[2].x - tr.p[0].x);
                                splitPoint.y = tr.p[1].y;
                                SDL_Point points[3];
                                points[0] = tr.p[0];
                                points[1] = tr.p[1];
                                points[2] = splitPoint;
                                DrawBottomFlatTriangle(renderer, points);
                                points[0] = tr.p[1];
                                points[1] = splitPoint;
                                points[2] = tr.p[2];
                                DrawTopFlatTriangle(renderer, points);
                          }
                    }
```

```
	}

	void FillTrianglesToRasterVector(std::vector<Triangle> &vecTrianglesToRaster, Triangle &tri, Matrix4 &matWorld, Matrix4 &matView) {
		Triangle triProjected, triTransformed, triViewed;

		// World Matrix Transform
		triTransformed.p[0] = Matrix4_MultiplyVector(tri.p[0], matWorld);
		triTransformed.p[1] = Matrix4_MultiplyVector(tri.p[1], matWorld);
		triTransformed.p[2] = Matrix4_MultiplyVector(tri.p[2], matWorld);
		/*triTransformed.R = tri.R;
		triTransformed.G = tri.G;
		triTransformed.B = tri.B;*/
		triTransformed.color = tri.color;

		// Calculate triangle Normal
		vec3 normal, line1, line2;

		// Get lines either side of triangle
		line1 = Vector3_Sub(triTransformed.p[1], triTransformed.p[0]);
		line2 = Vector3_Sub(triTransformed.p[2], triTransformed.p[0]);

		// Take cross product of lines to get normal to triangle surface
		normal = Vector3_CrossProduct(line1, line2);

		// You normally need to normalise a normal!
		normal = Vector3_Normalize(normal);

		// Get Ray from triangle to camera
		vec3 vCameraRay = Vector3_Sub(triTransformed.p[0], MainCamera.position);

		if (Vector3_DotProduct(normal, vCameraRay) < 0.0f) {

			// How similar is normal to light direction
			float dp = normal.x * LightDirection.x + normal.y * LightDirection.y + normal.z * LightDirection.z;
			if (dp < 0.1f) {
				dp = 0.1f;
			}
			triTransformed.color.R = dp * triTransformed.color.R;
			triTransformed.color.G = dp * triTransformed.color.G;
			triTransformed.color.B = dp * triTransformed.color.B;

			// Convert World Space --> View Space
			triViewed.p[0] = Matrix4_MultiplyVector(triTransformed.p[0], matView);
			triViewed.p[1] = Matrix4_MultiplyVector(triTransformed.p[1], matView);
			triViewed.p[2] = Matrix4_MultiplyVector(triTransformed.p[2], matView);
			/*triViewed.R = triTransformed.R;
			triViewed.G = triTransformed.G;
			triViewed.B = triTransformed.B;*/
			triViewed.color = triTransformed.color;

			int nClippedTriangles = 0;
			Triangle clipped[2];
			nClippedTriangles = Triangle_ClipAgainstPlane({ 0.0f, 0.0f, 0.1f }, { 0.0f, 0.0f, 1.0f }, triViewed, clipped[0], clipped[1]);

			for (int n = 0; n < nClippedTriangles; n++)
			{
				// Project triangles from 3D --> 2D
				triProjected.p[0] = Matrix4_MultiplyVector(clipped[n].p[0], matProj);
				triProjected.p[1] = Matrix4_MultiplyVector(clipped[n].p[1], matProj);
				triProjected.p[2] = Matrix4_MultiplyVector(clipped[n].p[2], matProj);
```

```
                                /*triProjected.R = clipped[n].R;
                                triProjected.G = clipped[n].G;
                                triProjected.B = clipped[n].B;*/
                                triProjected.color = clipped[n].color;

                                // X/Y are inverted so put them back
                                triProjected.p[0].x *= -1.0f;
                                triProjected.p[1].x *= -1.0f;
                                triProjected.p[2].x *= -1.0f;
                                triProjected.p[0].y *= -1.0f;
                                triProjected.p[1].y *= -1.0f;
                                triProjected.p[2].y *= -1.0f;

                                // Offset verts into visible normalised space
                                vec3 vOffsetView = { 1,1,0 };
                                triProjected.p[0] = Vector3_Add(triProjected.p[0], vOffsetView);
                                triProjected.p[1] = Vector3_Add(triProjected.p[1], vOffsetView);
                                triProjected.p[2] = Vector3_Add(triProjected.p[2], vOffsetView);
                                triProjected.p[0].x *= 0.5f * WIDTH;
                                triProjected.p[0].y *= 0.5f * HEIGHT;
                                triProjected.p[1].x *= 0.5f * WIDTH;
                                triProjected.p[1].y *= 0.5f * HEIGHT;
                                triProjected.p[2].x *= 0.5f * WIDTH;
                                triProjected.p[2].y *= 0.5f * HEIGHT;

                                // Store triangle for sorting
                                vecTrianglesToRaster.push_back(triProjected);
                        }
                }
        }

        void DrawSceneObjects(SDL_Renderer *renderer) {
                Matrix4 matRotX, matRotY, matRotZ;

                /*static float theta = 0;
                theta += 1.0f / 30.0f;*/

                // Rotation X
                matRotX = Matrix4_MakeRotationX(0);

                // Rotation Y
                matRotY = Matrix4_MakeRotationY(0);

                // Rotation Z
                matRotZ = Matrix4_MakeRotationZ(0);

                Matrix4 matTrans;
                matTrans = Matrix4_MakeTranslation(0.0f, 0.0f, 5.0f);

                Matrix4 matWorld;
                matWorld = Matrix4_MakeIdentity();          // Form World Matrix
                matWorld = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
                matWorld = Matrix4_MultiplyMatrix(matWorld, matRotZ); // Transform by rotation by Y
                matWorld = Matrix4_MultiplyMatrix(matWorld, matTrans); // Transform by translation

                vec3 upVector = { 0, 1, 0 };
                vec3 targetVector = { 0, 0, 1 };
Matrix4 m1 = Matrix4_MakeRotationX(MainCamera.fXRotation);
Matrix4 m2 = Matrix4_MakeRotationY(MainCamera.fYRotation);
                Matrix4 matCameraRot = Matrix4_MultiplyMatrix(m1, m2);
                MainCamera.lookDirection = Matrix4_MultiplyVector(targetVector, matCameraRot);
                targetVector = Vector3_Add(MainCamera.position, MainCamera.lookDirection);
```

```
		Matrix4 matCamera = Matrix4_PointAt(MainCamera.position, targetVector, upVector);

		Matrix4 matView = Matrix4_QuickInverse(matCamera);

		// Store triagles for rastering later
		std::vector<Triangle> vecTrianglesToRaster;

		for (GE_Object &obj : GE_DRAW_LIST.obj) {
			//if (!Vector3_Equals(obj.getPosition(), GE_DRAW_LIST.selectorBox.getPosition())) {
				for (Mesh_Side &side : obj.sides) {
					if (!side.hidden) {
						for (Triangle &tri : side.mesh.polygons) {
							FillTrianglesToRasterVector(vecTrianglesToRaster, tri, mat-
World, matView);
						}
					}
				}
			//}
		}

		for (Mesh_Side &side : GE_DRAW_LIST.selectorBox.sides) {
			if (!side.hidden) {
				for (Triangle &tri : side.mesh.polygons) {
					FillTrianglesToRasterVector(vecTrianglesToRaster, tri, matWorld, matView);
				}
			}
		}

		sort(vecTrianglesToRaster.begin(), vecTrianglesToRaster.end(), [](Triangle &t1, Triangle &t2)
			{
				float z1 = (t1.p[0].z + t1.p[1].z + t1.p[2].z) / 3.0f;
				float z2 = (t2.p[0].z + t2.p[1].z + t2.p[2].z) / 3.0f;
				return z1 > z2;
			});

		for (Triangle &triToRaster : vecTrianglesToRaster)
		{
			// Clip triangles against all four screen edges, this could yield
			// a bunch of triangles, so create a queue that we traverse to
			//  ensure we only test new triangles generated against planes
			Triangle clipped[2];
			std::list<Triangle> listTriangles;

			// Add initial triangle
			listTriangles.push_back(triToRaster);
			unsigned long nNewTriangles = 1;

			for (int p = 0; p < 4; p++)
			{
				int nTrisToAdd = 0;
				while (nNewTriangles > 0)
				{
					// Take triangle from front of queue
					Triangle test = listTriangles.front();
					listTriangles.pop_front();
					nNewTriangles--;

					// Clip it against a plane. We only need to test each
					// subsequent plane, against subsequent new triangles
					// as all triangles after a plane clip are guaranteed
					// to lie on the inside of the plane. I like how this
					// comment is almost completely and utterly justified
```

```
                                        switch (p)
                                        {
                                        case 0:   nTrisToAdd = Triangle_ClipAgainstPlane({ 0.0f, 0.0f, 0.0f }, { 0.0f, 1.0f,
0.0f }, test, clipped[0], clipped[1]); break;
                                        case 1:   nTrisToAdd = Triangle_ClipAgainstPlane({ 0.0f, (float)HEIGHT - 1.0f,
0.0f }, { 0.0f, -1.0f, 0.0f }, test, clipped[0], clipped[1]); break;
                                        case 2:   nTrisToAdd = Triangle_ClipAgainstPlane({ 0.0f, 0.0f, 0.0f }, { 1.0f, 0.0f,
0.0f }, test, clipped[0], clipped[1]); break;
                                        case 3:   nTrisToAdd = Triangle_ClipAgainstPlane({ (float)WIDTH - 1.0f, 0.0f,
0.0f }, { -1.0f, 0.0f, 0.0f }, test, clipped[0], clipped[1]); break;
                                        }

                                        // Clipping may yield a variable number of triangles, so
                                        // add these new ones to the back of the queue for subsequent
                                        // clipping against next planes
                                        for (int w = 0; w < nTrisToAdd; w++) {
                                                listTriangles.push_back(clipped[w]);
                                        }

                                }
                                nNewTriangles = listTriangles.size();
                        }


                        // Draw the transformed, viewed, clipped, projected, sorted, clipped triangles
                        for (Triangle &t : listTriangles)
                        {
                                SDL_Point points[3] = {
                                                        { (int)t.p[0].x,(int)t.p[0].y },
                                                        { (int)t.p[1].x,(int)t.p[1].y },
                                                        { (int)t.p[2].x,(int)t.p[2].y }
                                };
                                Triangle2D tr = { points[0], points[1], points[2] };
                                switch (GE_RENDERING_STYLE)
                                {
                                case Engine3D::RENDERING_STYLES::STD_SHADED:
                                        SDL_SetRenderDrawColor(renderer, t.color.R, t.color.G, t.color.B, 255.0f);
                                        DrawFilledTriangle2D(renderer, tr);
                                        break;
                                case Engine3D::RENDERING_STYLES::STD_POLY_SHADED:
                                        SDL_SetRenderDrawColor(renderer, t.color.R, t.color.G, t.color.B, 255.0f);
                                        DrawFilledTriangle2D(renderer, tr);
                                        SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
                                        DrawTriangle2D(renderer, tr);
                                        break;
                                case Engine3D::RENDERING_STYLES::DEBUG_DRAW_ONLY_POLYGONS:
                                        SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
                                        DrawTriangle2D(renderer, tr);
                                        break;
                                default:
                                        break;
                                }
                        }
                }

        }

        void resetMainCamera() {
                MainCamera.position = { 3, 4, 0 };
                MainCamera.lookDirection = { 0, 0, 1 };
                MainCamera.fXRotation = M_PI / 6.0f; //3.14159f / 1000.0f;
                MainCamera.fYRotation = M_PI / 8.0f; // 3.14159f / 4.0f;
```

```
		MainCamera.fFOV = 90.0f;
		MainCamera.fNear = 0.1f;
		MainCamera.fFar = 1000.0f;
	}

	GE_Object* getGEObjectPointerByPos(vec3 pos) {
		for (GE_Object &obj : GE_DRAW_LIST.obj) {
			if (Vector3_Equals(obj.getPosition(), pos)) {
				return &obj;
			}
		}
		return nullptr;
	}

	void ShowPreviouslyUnneededSidesByObj(GE_Object *obj) {
		struct vec3_wd {
			GE_MESH_SIDE_TYPE sideType = GE_MESH_SIDE_TYPE::UNDEFINED;
			GE_MESH_SIDE_TYPE counterSideType = GE_MESH_SIDE_TYPE::UNDEFINED;
			vec3 vec;
		};
		switch (obj->getObjType()) {
		case GE_OBJECT_TYPE::CUBE: {
			vec3 objPos = obj->getPosition();

			std::vector<vec3_wd> neighbourPositions;
			neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::EAST,
	GE_MESH_SIDE_TYPE::WEST,  1,  0,  0 });
			neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::TOP, GE_MESH_SIDE_TYPE::BOT-
TOM,    0,  1,  0 });
			neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::NORTH,
GE_MESH_SIDE_TYPE::SOUTH,           0,  0,  1 });
			neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::WEST,
GE_MESH_SIDE_TYPE::EAST,  -1,  0,  0 });
			neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::BOTTOM,
GE_MESH_SIDE_TYPE::TOP,     0, -1,  0 });
			neighbourPositions.push_back({
GE_MESH_SIDE_TYPE::SOUTH,GE_MESH_SIDE_TYPE::NORTH,          0,  0, -1 });
			for (vec3_wd &v_wd : neighbourPositions) {
				v_wd.vec = Vector3_Add(v_wd.vec, objPos);
				GE_Object *neighbourObj = getGEObjectPointerByPos(v_wd.vec);
				if (neighbourObj != nullptr) {
					switch (neighbourObj->getObjType()) {
					case GE_OBJECT_TYPE::CUBE: {
						neighbourObj->showSide(v_wd.counterSideType);
						break;
					}
					default:
						break;
					}
				}
			}

			break;
		}
		default:
			break;
		}
	}

	void HideUnneededSidesByObj(GE_Object *obj) {
		struct vec3_wd {
			GE_MESH_SIDE_TYPE sideType = GE_MESH_SIDE_TYPE::UNDEFINED;
```

```
                        GE_MESH_SIDE_TYPE counterSideType = GE_MESH_SIDE_TYPE::UNDEFINED;
                        vec3 vec;
                };
                switch (obj->getObjType()) {
                case GE_OBJECT_TYPE::CUBE: {
                        vec3 objPos = obj->getPosition();

                        std::vector<vec3_wd> neighbourPositions;
                        neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::EAST,
        GE_MESH_SIDE_TYPE::WEST,  1,  0,  0 });
                        neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::TOP, GE_MESH_SIDE_TYPE::BOT-
TOM,     0,  1,  0 });
                        neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::NORTH,
GE_MESH_SIDE_TYPE::SOUTH,             0,  0,  1 });
                        neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::WEST,
GE_MESH_SIDE_TYPE::EAST,  -1,  0,  0 });
                        neighbourPositions.push_back({ GE_MESH_SIDE_TYPE::BOTTOM,
GE_MESH_SIDE_TYPE::TOP,     0, -1,  0 });
                        neighbourPositions.push_back({
GE_MESH_SIDE_TYPE::SOUTH,GE_MESH_SIDE_TYPE::NORTH,           0,  0, -1 });
                        for (vec3_wd &v_wd : neighbourPositions) {
                                v_wd.vec = Vector3_Add(v_wd.vec, objPos);
                                GE_Object *neighbourObj = getGEObjectPointerByPos(v_wd.vec);
                                if (neighbourObj != nullptr) {
                                        switch (neighbourObj->getObjType()) {
                                        case GE_OBJECT_TYPE::CUBE: {
                                                neighbourObj->hideSide(v_wd.counterSideType);
                                                obj->hideSide(v_wd.sideType);
                                                break;
                                        }
                                        default:
                                                break;
                                        }
                                }
                        }

                        break;
                }
                default:
                        break;
                }
        }

        void CreateBlockAtSelectorPosition() {
                vec3 pos = GE_DRAW_LIST.selectorBox.getPosition();
                GE_Object *obj = getGEObjectPointerByPos(pos);
                if (obj == nullptr){
                        GE_Object sBox = GE_STD_OBJECTS.CUBE;
                        sBox.moveTo(pos);
                        HideUnneededSidesByObj(&sBox);
                        GE_DRAW_LIST.obj.push_back(sBox);
                        printf("Created block at %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
                } else {
                        printf("Aborted to created block at %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
                }
        }

        void RemoveBlockAtSelectorPosition() {
                // Can't use getGEObjectPointerByPos() because we need to remove from vector by index
                vec3 pos = GE_DRAW_LIST.selectorBox.getPosition();
                int index = -1;
                int i = 0;
```

```
	for (GE_Object &obj : GE_DRAW_LIST.obj) {
		if (Vector3_Equals(obj.getPosition(), pos)) {
			index = i;
			break;
		}
		i++;
	}
	if (index >= 0)	{
		ShowPreviouslyUnneededSidesByObj(&(GE_DRAW_LIST.obj[index]));
		GE_DRAW_LIST.obj.erase(GE_DRAW_LIST.obj.begin() + index);
		printf("Removed block at %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
	} else {
		printf("Unable to remove block at %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
	}
}

void ChangeBlockColorAtSelectorPosition() {
	vec3 pos = GE_DRAW_LIST.selectorBox.getPosition();
	GE_Object *obj = getGEObjectPointerByPos(pos);
	if (obj != nullptr) {
		Colors::Types newColorType = GE_COLORS.getColorTypeAfter(obj->getColorType());
		GE_Color newColor = GE_COLORS.getColorByType(newColorType);
		obj->setColor(newColor, newColorType);
		printf("Changed block color at %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
	}
	else {
		printf("Aborted to change block color at %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
	}
}

void SceneEditingHandle(SDL_Scancode scancode) {
	//printf("SceneEditingHandle->");
	switch (scancode) {
	case SDL_SCANCODE_UP: {
		//printf("Tapped SDL_SCANCODE_UP\n");
		GE_DRAW_LIST.selectorBox.moveBy({ 0, 0, 1 });
		break;
	}
	case SDL_SCANCODE_DOWN: {
		//printf("Tapped SDL_SCANCODE_DOWN\n");
		GE_DRAW_LIST.selectorBox.moveBy({ 0, 0, -1 });
		break;
	}
	case SDL_SCANCODE_LEFT: {
		//printf("Tapped SDL_SCANCODE_LEFT\n");
		GE_DRAW_LIST.selectorBox.moveBy({ 1, 0, 0 });
		break;
	}
	case SDL_SCANCODE_RIGHT: {
		//printf("Tapped SDL_SCANCODE_RIGHT\n");
		GE_DRAW_LIST.selectorBox.moveBy({ -1, 0, 0 });
		break;
	}
	case SDL_SCANCODE_SPACE: {
		//printf("Tapped SDL_SCANCODE_SPACE\n");
		GE_DRAW_LIST.selectorBox.moveBy({ 0, 1, 0 });
		break;
	}
	case SDL_SCANCODE_X: {
		//printf("Tapped SDL_SCANCODE_X\n");
		GE_DRAW_LIST.selectorBox.moveBy({ 0, -1, 0 });
		break;
```

```
		}
		case SDL_SCANCODE_F: {
			//printf("Tapped SDL_SCANCODE_F\n");
			CreateBlockAtSelectorPosition();
			break;
		}
		case SDL_SCANCODE_R: {
			//printf("Tapped SDL_SCANCODE_R\n");
			RemoveBlockAtSelectorPosition();
			break;
		}
		case SDL_SCANCODE_E: {
			//printf("Tapped SDL_SCANCODE_E\n");
			ChangeBlockColorAtSelectorPosition();
			break;
		}
		case SDL_SCANCODE_GRAVE: {
			//printf("Tapped SDL_SCANCODE_GRAVE\n");
			GE_CURRENT_KEYBOARD_CONTROL = Engine3D::KEYBOARD_CONTROL_TYPES::AL-
LOW_CAMERA_CONTROL;
			break;
		}
		default:
			printf("Tapped untreated key: %d\n", scancode);
			break;
		}
	}

	void CameraMovementHandle(SDL_Scancode scancode) {
		float offset = 20.0f / (float)FRAMES_PER_SECOND;
		//printf("CameraMovementHandle->");
		switch (scancode) {
			// WASD
		case SDL_SCANCODE_W: {
			//printf("Tapped SDL_SCANCODE_W\n");
			MainCamera.position.z += offset;
			break;
		}
		case SDL_SCANCODE_A: {
			//printf("Tapped SDL_SCANCODE_A\n");
			MainCamera.position.x += offset;
			break;
		}
		case SDL_SCANCODE_S: {
			//printf("Tapped SDL_SCANCODE_S\n");
			MainCamera.position.z -= offset;
			break;
		}
		case SDL_SCANCODE_D: {
			//printf("Tapped SDL_SCANCODE_D\n");
			MainCamera.position.x -= offset;
			break;
		}
		case SDL_SCANCODE_SPACE: {
			//printf("Tapped SDL_SCANCODE_SPACE\n");
			MainCamera.position.y += offset;
			break;
		}
		case SDL_SCANCODE_X: {
			//printf("Tapped SDL_SCANCODE_X\n");
			MainCamera.position.y -= offset;
			break;
```

```
		}
		case SDL_SCANCODE_UP: {
			//printf("Tapped SDL_SCANCODE_UP\n");
			MainCamera.fXRotation -= offset;
			break;
		}
		case SDL_SCANCODE_DOWN: {
			//printf("Tapped SDL_SCANCODE_DOWN\n");
			MainCamera.fXRotation += offset;
			break;
		}
		case SDL_SCANCODE_LEFT: {
			//printf("Tapped SDL_SCANCODE_LEFT\n");
			MainCamera.fYRotation -= offset;
			break;
		}
		case SDL_SCANCODE_RIGHT: {
			//printf("Tapped SDL_SCANCODE_RIGHT\n");
			MainCamera.fYRotation += offset;
			break;
		}
		case SDL_SCANCODE_RIGHTBRACKET: {
			//printf("Tapped SDL_SCANCODE_RIGHTBRACKET\n");
			if (MainCamera.fFOV > 0 + 1)
			{
				MainCamera.fFOV--;
			}
			printf("%f\n", MainCamera.fFOV);
			break;
		}
		case SDL_SCANCODE_LEFTBRACKET: {
			//printf("Tapped SDL_SCANCODE_LEFTBRACKET\n");
			if (MainCamera.fFOV < 180 - 1)
			{
				MainCamera.fFOV++;
			}
			printf("%f\n", MainCamera.fFOV);
			break;
		}
		case SDL_SCANCODE_R: {
			//printf("Tapped SDL_SCANCODE_R\n");
			resetMainCamera();
			break;
		}
		case SDL_SCANCODE_GRAVE: {
			//printf("Tapped SDL_SCANCODE_GRAVE\n");
			GE_CURRENT_KEYBOARD_CONTROL = Engine3D::KEYBOARD_CONTROL_TYPES::AL-
LOW_SCENE_EDITING;
			break;
		}
		default:
			printf("Tapped untreated key: %d\n", scancode);
			break;
		}
	}

	void StartRenderLoop() {
		SDL_Renderer *renderer = SDL_CreateRenderer(window, -1, 0);
		Uint32 start;
		SDL_Event windowEvent;
		while (isRunning)
		{
```

```
                        if (SDL_PollEvent(&windowEvent))
                        {
                                if (windowEvent.type == SDL_QUIT)
                                {
                                        isRunning = false;
                                }

                                if (windowEvent.type == SDL_KEYDOWN) {
                                        switch (GE_CURRENT_KEYBOARD_CONTROL) {
                                        case Engine3D::KEYBOARD_CONTROL_TYPES::ALLOW_SCENE_EDIT-
ING:
                                                SceneEditingHandle(windowEvent.key.keysym.scancode);
                                                break;
                                        case Engine3D::KEYBOARD_CONTROL_TYPES::ALLOW_OBJECT_EDIT-
ING:
                                                break;
                                        case Engine3D::KEYBOARD_CONTROL_TYPES::ALLOW_CAMERA_CON-
TROL:
                                                CameraMovementHandle(windowEvent.key.keysym.scancode);
                                                break;
                                        default:
                                                break;
                                        }
                                }
                        }
                        start = SDL_GetTicks();

                        //Updates properties of the screen and camera
                        updateScreenAndCameraProperties(renderer);

                        //Background(Clears with color)
                        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255);
                        SDL_RenderClear(renderer);

                        //Draws scene
                        SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
                        DrawSceneObjects(renderer);

                        // Renders window
                        SDL_RenderPresent(renderer);
                        Uint32 ticks = SDL_GetTicks();
                        if (1000 / FRAMES_PER_SECOND > ticks - start) {
                                SDL_Delay(1000 / FRAMES_PER_SECOND - (ticks - start));
                        }
                }
                SDL_DestroyRenderer(renderer);
        }

        void CreateCubicFormByTopMesh(GE_Object &buffObj, Mesh &Mesh_TOP) {
                Mesh_Side buffSide;

                Matrix4 matRotX;
                Matrix4 matRotY;
                Matrix4 matRotZ;
                Matrix4 matRot;

                buffSide.type = GE_MESH_SIDE_TYPE::TOP;
                buffSide.mesh = Mesh_TOP;
                matRotX = Matrix4_MakeRotationX(0);
                matRotY = Matrix4_MakeRotationY(0);
                matRotZ = Matrix4_MakeRotationZ(0);
                matRot = Matrix4_MakeIdentity();
```

```
matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
for (Triangle &polygon : buffSide.mesh.polygons) {
	for (vec3 &v : polygon.p) {
		v = Matrix4_MultiplyVector(v, matRot);
	}
}
buffObj.sides.push_back(buffSide);


buffSide.type = GE_MESH_SIDE_TYPE::BOTTOM;
buffSide.mesh = Mesh_TOP;
matRotX = Matrix4_MakeRotationX(0);
matRotY = Matrix4_MakeRotationY(0);
matRotZ = Matrix4_MakeRotationZ(M_PI);
matRot = Matrix4_MakeIdentity();
matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
for (Triangle &polygon : buffSide.mesh.polygons) {
	for (vec3 &v : polygon.p) {
		v = Matrix4_MultiplyVector(v, matRot);
	}
}
buffObj.sides.push_back(buffSide);

buffSide.type = GE_MESH_SIDE_TYPE::WEST;
buffSide.mesh = Mesh_TOP;
matRotX = Matrix4_MakeRotationX(0);
matRotY = Matrix4_MakeRotationY(0);
matRotZ = Matrix4_MakeRotationZ(M_PI / 2.0f);
matRot = Matrix4_MakeIdentity();
matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
for (Triangle &polygon : buffSide.mesh.polygons) {
	for (vec3 &v : polygon.p) {
		v = Matrix4_MultiplyVector(v, matRot);
	}
}
buffObj.sides.push_back(buffSide);

buffSide.type = GE_MESH_SIDE_TYPE::EAST;
buffSide.mesh = Mesh_TOP;
matRotX = Matrix4_MakeRotationX(0);
matRotY = Matrix4_MakeRotationY(0);
matRotZ = Matrix4_MakeRotationZ(-M_PI / 2.0f);
matRot = Matrix4_MakeIdentity();
matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
for (Triangle &polygon : buffSide.mesh.polygons) {
	for (vec3 &v : polygon.p) {
		v = Matrix4_MultiplyVector(v, matRot);
	}
}
buffObj.sides.push_back(buffSide);

buffSide.type = GE_MESH_SIDE_TYPE::NORTH;
buffSide.mesh = Mesh_TOP;
matRotX = Matrix4_MakeRotationX(M_PI / 2.0f);
matRotY = Matrix4_MakeRotationY(0);
matRotZ = Matrix4_MakeRotationZ(0);
matRot = Matrix4_MakeIdentity();
matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
```

```
		matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
		for (Triangle &polygon : buffSide.mesh.polygons) {
			for (vec3 &v : polygon.p) {
				v = Matrix4_MultiplyVector(v, matRot);
			}
		}
		buffObj.sides.push_back(buffSide);

		buffSide.type = GE_MESH_SIDE_TYPE::SOUTH;
		buffSide.mesh = Mesh_TOP;
		matRotX = Matrix4_MakeRotationX(-M_PI / 2.0f);
		matRotY = Matrix4_MakeRotationY(0);
		matRotZ = Matrix4_MakeRotationZ(0);
		matRot = Matrix4_MakeIdentity();
		matRot = Matrix4_MultiplyMatrix(matRotX, matRotY); // Transform by rotation by X and Y
		matRot = Matrix4_MultiplyMatrix(matRot, matRotZ); // Transform by rotation by Y
		for (Triangle &polygon : buffSide.mesh.polygons) {
			for (vec3 &v : polygon.p) {
				v = Matrix4_MultiplyVector(v, matRot);
			}
		}
		buffObj.sides.push_back(buffSide);
}

void initStdSelector() {
		GE_Object buffObj;
		buffObj.setObjType(GE_OBJECT_TYPE::SELECTOR);

		Mesh Mesh_TOP;
		Mesh_TOP.polygons = {
			// OUTER PART
			{  1.0f, 1.2f, 0.0f,   1.0f, 1.2f,  0.2f,   1.2f, 1.2f,  0.2f },
			{  1.2f, 1.2f, 0.2f,   1.2f, 1.2f, -0.2f,   0.8f, 1.2f, -0.2f },
			{  1.0f, 1.2f, 0.0f,   0.8f, 1.2f, -0.2f,   0.8f, 1.2f,  0.0f },

			{  0.0f, 1.2f, 0.0f,  -0.2f, 1.2f,  0.2f,   0.0f, 1.2f,  0.2f },
			{ -0.2f, 1.2f, 0.2f,   0.2f, 1.2f, -0.2f,  -0.2f, 1.2f, -0.2f },
			{  0.0f, 1.2f, 0.0f,   0.2f, 1.2f,  0.0f,   0.2f, 1.2f, -0.2f },

			{  0.0f, 1.2f, 0.8f,  -0.2f, 1.2f,  0.8f,   0.0f, 1.2f,  1.0f },
			{ -0.2f, 1.2f, 0.8f,  -0.2f, 1.2f,  1.2f,   0.2f, 1.2f,  1.2f },
			{  0.2f, 1.2f, 1.2f,   0.2f, 1.2f,  1.0f,   0.0f, 1.2f,  1.0f },

			{  1.0f, 1.2f, 1.0f,   0.8f, 1.2f,  1.0f,   0.8f, 1.2f,  1.2f },
			{  0.8f, 1.2f, 1.2f,   1.2f, 1.2f,  1.2f,   1.2f, 1.2f,  0.8f },
			{  1.2f, 1.2f, 0.8f,   1.0f, 1.2f,  0.8f,   1.0f, 1.2f,  1.0f },

			// INNER PART
			{  1.0f, 0.2f,  0.0f,   1.2f, 0.2f,  0.0f,   1.0f, 0.2f, -0.2f },
			{  1.0f, 0.2f, -0.2f,   1.2f, 0.2f,  0.0f,   1.2f, 0.2f, -0.2f },

			{  -0.2f, 0.2f,  0.0f,   0.0f, 0.2f,  0.0f,   -0.2f, 0.2f, -0.2f },
			{  -0.2f, 0.2f, -0.2f,   0.0f, 0.2f,  0.0f,    0.0f, 0.2f, -0.2f },

			{  -0.2f, 0.2f,  1.2f,   0.0f, 0.2f,  1.2f,   -0.2f, 0.2f, 1.0f },
			{  -0.2f, 0.2f,  1.0f,   0.0f, 0.2f,  1.2f,    0.0f, 0.2f, 1.0f },

			{  1.0f, 0.2f,  1.2f,   1.2f, 0.2f,  1.2f,   1.0f, 0.2f, 1.0f },
			{  1.0f, 0.2f,  1.0f,   1.2f, 0.2f,  1.2f,   1.2f, 0.2f, 1.0f },


			{  0.8f, 0.0f,  0.0f,   1.0f, 0.0f,  0.0f,   0.8f, 0.0f, -0.2f },
```

```
                { 0.8f, 0.0f, -0.2f,   1.0f, 0.0f,  0.0f,   1.0f, 0.0f, -0.2f },

                { 0.0f, 0.0f,  0.0f,   0.2f, 0.0f,  0.0f,   0.0f, 0.0f, -0.2f },
                { 0.0f, 0.0f, -0.2f,   0.2f, 0.0f,  0.0f,   0.2f, 0.0f, -0.2f },

                { 0.0f, 0.0f,  1.2f,   0.2f, 0.0f,  1.2f,   0.0f, 0.0f, 1.0f },
                { 0.0f, 0.0f,  1.0f,   0.2f, 0.0f,  1.2f,   0.2f, 0.0f, 1.0f },

                { 0.8f, 0.0f,  1.2f,   1.0f, 0.0f,  1.2f,   0.8f, 0.0f, 1.0f },
                { 0.8f, 0.0f,  1.0f,   1.0f, 0.0f,  1.2f,   1.0f, 0.0f, 1.0f },


                { 1.0f, 0.0f,  0.2f,   1.2f, 0.0f,  0.2f,   1.0f, 0.0f, 0.0f },
                { 1.0f, 0.0f,  0.0f,   1.2f, 0.0f,  0.2f,   1.2f, 0.0f, 0.0f },

                { -0.2f, 0.0f,  0.2f,   0.0f, 0.0f,  0.2f,   -0.2f, 0.0f, 0.0f },
                { -0.2f, 0.0f,  0.0f,   0.0f, 0.0f,  0.2f,    0.0f, 0.0f, 0.0f },

                { -0.2f, 0.0f,  1.0f,   0.0f, 0.0f,  1.0f,   -0.2f, 0.0f, 0.8f },
                { -0.2f, 0.0f,  0.8f,   0.0f, 0.0f,  1.0f,    0.0f, 0.0f, 0.8f },

                { 1.0f, 0.0f,  1.0f,   1.2f, 0.0f,  1.0f,   1.0f, 0.0f, 0.8f },
                { 1.0f, 0.0f,  0.8f,   1.2f, 0.0f,  1.0f,   1.2f, 0.0f, 0.8f },
        };

        for (Triangle &polygon : Mesh_TOP.polygons) {
                for (vec3 &v : polygon.p) {
                        v = Vector3_Add(v, ObjectMeshAnchor);
                }
        }

        CreateCubicFormByTopMesh(buffObj, Mesh_TOP);

        GE_STD_OBJECTS.SELECTOR = buffObj;
}

void initStdCube() {
        GE_Object buffObj;
        buffObj.setObjType(GE_OBJECT_TYPE::CUBE);

        Mesh Mesh_TOP;
        Mesh_TOP.polygons = {
                        { 0.0f, 1.0f, 0.0f,    0.0f, 1.0f, 1.0f,    1.0f, 1.0f, 1.0f },
                        { 0.0f, 1.0f, 0.0f,    1.0f, 1.0f, 1.0f,    1.0f, 1.0f, 0.0f },
        };
        for (Triangle &polygon : Mesh_TOP.polygons) {
                for (vec3 &v : polygon.p) {
                        v = Vector3_Add(v, ObjectMeshAnchor);
                }
        }

        CreateCubicFormByTopMesh(buffObj, Mesh_TOP);

        GE_STD_OBJECTS.CUBE = buffObj;
}

void initStdObjects() {
        initStdSelector();
        initStdCube();
}

void initSelectorObject() {
```

```
		GE_Object sBox = GE_STD_OBJECTS.SELECTOR;
		sBox.setColor(GE_COLORS.getColorByType(Colors::Types::YELLOW), Colors::Types::YELLOW);
		GE_DRAW_LIST.selectorBox = sBox;
	}

	ERROR_CODES initEngine() {
		if (SDL_Init(SDL_INIT_VIDEO) != 0) {
			return ERROR_CODES::SDL2_INIT_ERROR;
		}

		window = SDL_CreateWindow(
									title,
									SDL_WINDOWPOS_UNDEFINED,
									SDL_WINDOWPOS_UNDEFINED,
									WIDTH,
									HEIGHT,
									SDL_WINDOW_ALLOW_HIGHDPI |
SDL_WINDOW_RESIZABLE
								);

		// Check that the window was successfully created
		if (!check_window(window)) {
			return ERROR_CODES::WINDOW_INIT_ERROR;
		};

		initStdObjects();
		printf("Initialized std objects!\n");

		initSelectorObject();
		printf("Initialized selector object!\n");

		return ERROR_CODES::ZERO;
	}

	void Destroy() {
		SDL_DestroyWindow(window);
		SDL_Quit();
		printf("Destroyed 3DGE!\n");
	}

public:
	Engine3D(int _WIDTH, int _HEIGHT) {
		WIDTH = _WIDTH;
		HEIGHT = _HEIGHT;
		GE_ERROR_CODE = initEngine();
	}
	Engine3D() {
		WIDTH = 800;
		HEIGHT = 600;
		GE_ERROR_CODE = initEngine();
	}

	~Engine3D() {
		Destroy();
	}

	void startScene() {
		if (GE_ERROR_CODE != ERROR_CODES::ZERO) {
			printf("Unable to start scene! ERROR: %d\n", GE_ERROR_CODE);
			return;
		}
		resetMainCamera();
```

```
                printf("Reset camera!\n");
                isRunning = true;
                StartRenderLoop();
        }
};
```

```
int main(int argc, char *argv[]) {
        Engine3D Engine(800, 600);
        Engine.startScene();
   return 0;
}
```

```
#ifndef GE_3DMATH_H
#define GE_3DMATH_H

#include <math.h>

struct vec3 {
        float x, y, z;
};

struct Matrix4 {
        float m[4][4] = { 0 };
};

vec3 Vector3_Add(vec3 &v1, vec3 &v2) {
        return { v1.x + v2.x, v1.y + v2.y, v1.z + v2.z };
}

vec3 Vector3_Sub(vec3 &v1, vec3 &v2) {
        return { v1.x - v2.x, v1.y - v2.y, v1.z - v2.z };
}

vec3 Vector3_Mul(vec3 &v1, float k) {
        return { v1.x * k, v1.y * k, v1.z * k };
}

vec3 Vector3_Div(vec3 &v1, float k) {
        return { v1.x / k, v1.y / k, v1.z / k };
}

float Vector3_DotProduct(vec3 &v1, vec3 &v2) {
        return (v1.x * v2.x + v1.y * v2.y + v1.z * v2.z);
}

float Vector3_Length(vec3 &v)
{
        return sqrtf(Vector3_DotProduct(v, v));
}

bool Vector3_Equals(vec3 &v1, vec3 &v2) {
        if (v1.x == v2.x && v1.y == v2.y && v1.z == v2.z){
                return true;
        }
        return false;
```

```
}

vec3 Vector3_Normalize(vec3 &v) {
	float l = Vector3_Length(v);
	return { v.x / l, v.y / l, v.z / l };
}

vec3 Vector3_CrossProduct(vec3 &v1, vec3 &v2)
{
	vec3 v;
	v.x = v1.y * v2.z - v1.z * v2.y;
	v.y = v1.z * v2.x - v1.x * v2.z;
	v.z = v1.x * v2.y - v1.y * v2.x;
	return v;
}

vec3 Vector3_IntersectPlane(vec3 &plane_p, vec3 &plane_n, vec3 &lineStart, vec3 &lineEnd)
{
	plane_n = Vector3_Normalize(plane_n);
	float plane_d = -Vector3_DotProduct(plane_n, plane_p);
	float ad = Vector3_DotProduct(lineStart, plane_n);
	float bd = Vector3_DotProduct(lineEnd, plane_n);
	float t = (-plane_d - ad) / (bd - ad);
	vec3 lineStartToEnd = Vector3_Sub(lineEnd, lineStart);
	vec3 lineToIntersect = Vector3_Mul(lineStartToEnd, t);
	return Vector3_Add(lineStart, lineToIntersect);
}

vec3 Matrix4_MultiplyVector(vec3 &i, Matrix4 &m) {
	vec3 o;
	o.x = i.x * m.m[0][0] + i.y * m.m[1][0] + i.z * m.m[2][0] + m.m[3][0];
	o.y = i.x * m.m[0][1] + i.y * m.m[1][1] + i.z * m.m[2][1] + m.m[3][1];
	o.z = i.x * m.m[0][2] + i.y * m.m[1][2] + i.z * m.m[2][2] + m.m[3][2];

	float w = i.x * m.m[0][3] + i.y * m.m[1][3] + i.z * m.m[2][3] + m.m[3][3];
	if (w != 0.0f)
	{
		o.x /= w;
		o.y /= w;
		o.z /= w;
	}
	return o;
}

Matrix4 Matrix4_MakeIdentity() {
	Matrix4 matrix;
	matrix.m[0][0] = 1.0f;
	matrix.m[1][1] = 1.0f;
	matrix.m[2][2] = 1.0f;
	matrix.m[3][3] = 1.0f;
	return matrix;
}

Matrix4 Matrix4_MakeRotationX(float fAngleRad)
{
	Matrix4 matrix;
	matrix.m[0][0] = 1.0f;
	matrix.m[1][1] = cosf(fAngleRad);
	matrix.m[1][2] = sinf(fAngleRad);
	matrix.m[2][1] = -sinf(fAngleRad);
	matrix.m[2][2] = cosf(fAngleRad);
	matrix.m[3][3] = 1.0f;
	return matrix;
}
```

```
Matrix4 Matrix4_MakeRotationY(float fAngleRad)
{
	Matrix4 matrix;
	matrix.m[0][0] = cosf(fAngleRad);
	matrix.m[0][2] = sinf(fAngleRad);
	matrix.m[2][0] = -sinf(fAngleRad);
	matrix.m[1][1] = 1.0f;
	matrix.m[2][2] = cosf(fAngleRad);
	matrix.m[3][3] = 1.0f;
	return matrix;
}

Matrix4 Matrix4_MakeRotationZ(float fAngleRad)
{
	Matrix4 matrix;
	matrix.m[0][0] = cosf(fAngleRad);
	matrix.m[0][1] = sinf(fAngleRad);
	matrix.m[1][0] = -sinf(fAngleRad);
	matrix.m[1][1] = cosf(fAngleRad);
	matrix.m[2][2] = 1.0f;
	matrix.m[3][3] = 1.0f;
	return matrix;
}

Matrix4 Matrix4_MakeTranslation(float x, float y, float z)
{
	Matrix4 matrix;
	matrix.m[0][0] = 1.0f;
	matrix.m[1][1] = 1.0f;
	matrix.m[2][2] = 1.0f;
	matrix.m[3][3] = 1.0f;
	matrix.m[3][0] = x;
	matrix.m[3][1] = y;
	matrix.m[3][2] = z;
	return matrix;
}

Matrix4 Matrix4_MakeProjection(float fFovDegrees, float fAspectRatio, float fNear, float fFar)
{
	float fFovRad = 1.0f / tanf(fFovDegrees * 0.5f / 180.0f * 3.14159f);
	Matrix4 matrix;
	matrix.m[0][0] = fAspectRatio * fFovRad;
	matrix.m[1][1] = fFovRad;
	matrix.m[2][2] = fFar / (fFar - fNear);
	matrix.m[3][2] = (-fFar * fNear) / (fFar - fNear);
	matrix.m[2][3] = 1.0f;
	matrix.m[3][3] = 0.0f;
	return matrix;
}

Matrix4 Matrix4_MultiplyMatrix(Matrix4 &m1, Matrix4 &m2)
{
	Matrix4 matrix;
	for (int c = 0; c < 4; c++) {
		for (int r = 0; r < 4; r++) {
			matrix.m[r][c] = m1.m[r][0] * m2.m[0][c] + m1.m[r][1] * m2.m[1][c] +
m1.m[r][2] * m2.m[2][c] + m1.m[r][3] * m2.m[3][c];
		}
	}
	return matrix;
}

Matrix4 Matrix4_PointAt(vec3 &pos, vec3 &target, vec3 &up)
```

```
{
	// Calculate new forward direction
	vec3 newForward = Vector3_Sub(target, pos);
	newForward = Vector3_Normalize(newForward);

	// Calculate new Up direction
	vec3 a = Vector3_Mul(newForward, Vector3_DotProduct(up, newForward));
	vec3 newUp = Vector3_Sub(up, a);
	newUp = Vector3_Normalize(newUp);

	// New Right direction is easy, its just cross product
	vec3 newRight = Vector3_CrossProduct(newUp, newForward);

	// Construct Dimensioning and Translation Matrix
	Matrix4 matrix;
	matrix.m[0][0] = newRight.x;	matrix.m[0][1] = newRight.y;	matrix.m[0][2] =
newRight.z;	matrix.m[0][3] = 0.0f;
	matrix.m[1][0] = newUp.x;	matrix.m[1][1] = newUp.y;	matrix.m[1][2] =
newUp.z;		matrix.m[1][3] = 0.0f;
	matrix.m[2][0] = newForward.x;	matrix.m[2][1] = newForward.y;	matrix.m[2][2] = new-
Forward.z;	matrix.m[2][3] = 0.0f;
	matrix.m[3][0] = pos.x;		matrix.m[3][1] = pos.y;		ma-
trix.m[3][2] = pos.z;		matrix.m[3][3] = 1.0f;
	return matrix;

}

Matrix4 Matrix4_QuickInverse(Matrix4 &m) // Only for Rotation/Translation Matrixes
{
	Matrix4 matrix;
	matrix.m[0][0] = m.m[0][0]; matrix.m[0][1] = m.m[1][0]; matrix.m[0][2] = m.m[2][0]; ma-
trix.m[0][3] = 0.0f;
	matrix.m[1][0] = m.m[0][1]; matrix.m[1][1] = m.m[1][1]; matrix.m[1][2] = m.m[2][1]; ma-
trix.m[1][3] = 0.0f;
	matrix.m[2][0] = m.m[0][2]; matrix.m[2][1] = m.m[1][2]; matrix.m[2][2] = m.m[2][2]; ma-
trix.m[2][3] = 0.0f;
	matrix.m[3][0] = -(m.m[3][0] * matrix.m[0][0] + m.m[3][1] * matrix.m[1][0] + m.m[3][2] *
matrix.m[2][0]);
	matrix.m[3][1] = -(m.m[3][0] * matrix.m[0][1] + m.m[3][1] * matrix.m[1][1] + m.m[3][2] *
matrix.m[2][1]);
	matrix.m[3][2] = -(m.m[3][0] * matrix.m[0][2] + m.m[3][1] * matrix.m[1][2] + m.m[3][2] *
matrix.m[2][2]);
	matrix.m[3][3] = 1.0f;
	return matrix;
}

#endif
```