

Задание №3: Программная реализация графического конвейера и графического API

Владимир Фролов

4 октября 2025 г.

Аннотация

Цель задания: закрепить на практике основные работы графического конвейера через его реализацию.

- Закрепление знаний о трёхмерных преобразованиях и базовых математических операциях.
- Изучение базовых принципов работы графического конвейера;
- (Дополнительно) Программируемая функциональность граф-конвейера.
- (Дополнительно) Карты теней

1 Базовая часть: 10 баллов

Необходимо реализовать базовый алгоритм растеризации 3D моделей с поддержкой текстурирования.

- Необходимо реализовать заливку треугольника константным цветом (1 балл).
- Необходимо реализовать интерполяцию цвета (2 балла).
- Необходимо реализовать интерполяцию текстурных координат с перспективной коррекцией (3 балла).
- Необходимо реализовать алгоритм буфера глубины для корректного отображения близких и дальних объектов (4 балла).
- Для каждого изображения необходимо вывести время его рендеринга. При невыполнении этого требования баллы за базовую часть будут снижены вдвое!
- К заданию прилагается эталонная реализация через OpenGL 1.0. Для успешного выполнения задания необходимо получить хотя бы похожие (а в идеале совпадающие) с эталонными изображения на всех 8 тестах (рис. 1). При невыполнении этого требования баллы за базовую часть будут снижены на усмотрение проверяющего.

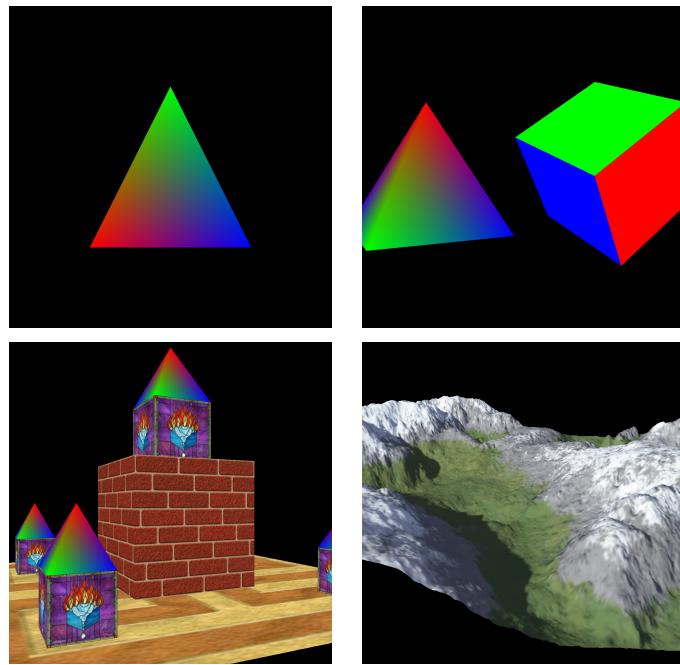


Рис. 1: Изображения некоторых тестовых сцен

2 Дополнительная часть: 15 баллов макс.

- Синтез последовательности изображений, содержащих поворот сцены вокруг центра координат или вращение некоторой 3D модели вокруг её центра (+1 балл).
 - Используйте ffmpeg чтобы получить видео-последовательность из набора изображений:


```
ffmpeg -framerate 25 -pattern_type glob -i '*.bmp' out.gif
```
 - В качестве альтернативного решения Вы можете выводить изображений в окно (проверка вашего задания будет осуществляться на Linux с оконной системой X11, как в примере для эталонной реализации). В этом случае пожалуйста выведите частоту кадров в название окна или в командную строку.
- Реализация прозрачности при помощи альфа-смешивания (2 балла для выпуклых объектов и 3 балла для не-выпуклых).
 - Добавьте новый режим и (при необходимости) новые поля в структуру PipelineStateObject
 - Добавьте режим отбраковки задних или передних поверхностей (по порядку обхода вершин – по часовой стрелки или против). Этот необходимо сделать т.к. для корректного отображения прозрачных объектов Вам сначала придётся рисовать задние грани, а затем передние

- Для не-выпуклых объектов необходимо реализовывать сортировку граней по расстоянию от камеры (координата z в пространстве камеры после применения worldView матрицы).
- Загрузка и рендеринг “.obj” файлов (2 балла).
 - Необходимо реализовать визуализацию как минимум 2 разных моделей
 - Вы можете использовать и другой входной формат 3D моделей.
- Реализация фрагментных шейдеров через указатели на функции (или аналогичную функциональность вашего языка программирования) (3 балла).
 - Добавьте необходимую входную информацию в структуру PipelineStateObject;
 - Необходимо реализовать рассчёт освещения во фрагментном шейдере (без этого пункта не засчитывается).
- Блочный алгоритм растеризации (2 балла).
- Применение векторизацию кода для блочного растеризатора и фрагментных шейдеров при помощи Intel ISPC (3 балла).
 - В этом пункте необходимо реализовать рассчёт освещения во фрагментном шейдере (без этого пункта не засчитывается).
 - Все векторизованные шейдеры должны быть собраны в объектные файлы и статически прилинкованы к проекты.
 - Необходимо привести сравнение времени выполнения для обычных и векторизованных шейдеров.
- Необходимо реализовать алгоритм карт теней. Необходимо продемонстрировать самозатенение или отбрасывание теней с одного 3D объекта на другие (4 балла).
- Растеризация квад-мешей (+3 балла).
 - В последнее время стали популярны меши, состоящие из четырёхугольников [2].
Преобразование таких мешей в треугольные снижает эффективность блочных алгоритмов растеризации на границе двух треугольников, образующих квад. Необходимо реализовать алгоритм растеризации квадов и сравнить время работы с растеризацией треугольников.
 - Существенно-непланарные квады необходимо всё-равно разбивать на треугольники.
 - Данный пункт имеет смысл выполнять только если Вы реализуете блочный векторизованный алгоритм.
 - Подготовленные для задания квад-меши можно найти здесь: [3].

3 Порядок выполнения задания

1. Скачайте или склонируйте себе шаблон для выполнения задания [1]
 - Пожалуйста не выкладывайте Ваше **решение** в открытый доступ. Дайте возможность другим людям достичь результата самостоятельно.
2. Соберите шаблон с эталонной реализацией и сгенерируйте эталонные изображения
 - `mkdir build && cd build`
 - `cmake -DCMAKE_BUILD_TYPE=Release (или Debug) -DUSE_OPENGL=ON ..`
 - `make -j 8`
 - Запустите полученную программу
3. Отключите эталонную реализацию и приступайте к выполнению задания
 - `cmake -DCMAKE_BUILD_TYPE=Release (или Debug) -DUSE_OPENGL=OFF ..`
 - `make -j 8`
4. Эталонная реализация создаёт окно при помощи X11, поэтому если в Вашей системе это невозможно, попросите товарища сгенерировать эталонные изображения для Вас.
5. Если Вы хотите выполнить задание на другом языке программирования, это допускается при условии совпадения вашей реализации с эталонными изображениями.
6. Изучите раздел 6 (Материалы для выполнения задания) и сайт [4].

4 Порядок сдачи

Ваша программа должна запускаться без аргументов, производить рендеринг изображений и сохранить их на выходе в bmp формат. Если Вы создаёте окно, то примите во внимание что проверка будет производиться на машинах Linux и оконной системой X11 (XWindow).

Использование библиотек:

Все используемые библиотеки должны быть либо собраны в статические библиотеки для архитектуры x64 с поддержкой AVX2 (-mavx2) под gcc, либо должны поставляться с проектом в виде исходных кодов, и фактически быть его частью. PS: строго говоря, для выполнения данного задания Вам не нужны библиотеки. Рекомендуется выполнять задание без них.

5 Именование архива

Архив, который вы загружаете на сайт должен называться по следующему шаблону:
<номер_группы>_<фамилия>.zip. Фамилию следует писать латинскими символами.

6 Материалы для выполнения задания

Итак, Вам выдали это скучнейшее в мире задание с растеризацией треугольников, в котором зачем-то требуется совпадение Вашей реализации с эталонной, написанной на умершем уже 10 лет как OpenGL 1.0. Что-ж, на практике Вам действительно придётся использовать другие API, шейдеры, GPU и многое другое. Поэтому совершенно точно можно сказать, что выполнив это задание Вы не получите практических навыков. Зато Вы получите громадный буст к их освоению, поскольку будете понимать как графические API, граф. конвейер (и даже современные GPU) устроены изнутри. Ведь в действительности там всё не так сложно, как кажется на первый взгляд.

6.1 Вопрос дизайна графического API

Современные графические API прошли достаточно длинный и не всегда эволюционный (а скорее наоборот) путь развития. Поэтому вопрос их дизайна не такой простой как может показаться на первый взгляд. Вам предлагается реализовать учебный GAPI, в котором функциональность урезана до минимума. Если он Вам не нравится, Вы можете придумать свой API и реализовать предложенный через него или переделать предложенные примеры на свой вариант интерфейса. Опишем требования, которые определяют дизайн нашего учебного API и обсудим особенности дизайна:

1. В базовой части задания необходимо отрисовывать непосредственно цвет в формате RGB, причём в двух режимах.
 - (a) Цвет задаётся в вершинах 3D модели
 - (b) Цвет задаётся текстурой, а в вершинах 3D модели заданы текстурные координаты
 - (c) В дополнительной части задания Вы можете расширить API для задания программируемой функциональностью, прозрачностью и др.
2. Режим отрисовки задаётся структурой PipelineStateObject. Такая структура так или иначе есть во всех графических API. Она полностью определяет то, как именно производится отрисовка определённой 3D модели и может содержать внутри себя ссылки на другие структуры (юниформы, шейдеры, состояния растеризатора и т.п.). Обратите внимание, что помимо режима отрисовки и идентификатора текстуры мы добавили в неё две матрицы – worldViewMatrix и projMatrix. Можно было бы отделить

матрицы в отдельную структуру (назовём её Uniforms), но на самом деле это не влияет на суть дела, поскольку в конечном итоге для правильной отрисовки необходимо иметь всю информацию о состоянии граф. конвейера, в том числе и юниформы.

3. Структура Geom задаёт саму 3D модель. Вам не нужно поддерживать различные способы (лэйауты, форматы) хранения геометрии. В данном задании геометрия хранится в одном, строго фиксированном представлении.
4. Инстансинг. Обычно для того чтобы отрисовать одну 3D модель несколько раз с разными 3D преобразованиями в GAPI добавляют специальные функции (вроде `glDrawElementsIndexedInstanced`). В нашем случае мы делаем это более явно, изменяя `PipelineStateObject` для каждого вызова `Draw` (с.м. функцию `DrawInstances`). То есть явной поддержки инстансинга в нашем GAPI нет. Тем не менее исходно геометрия организована таким образом, чтобы её можно было отрисовывать с инстансингом, поэтому при желании Вы можете добавить поддержку этой функциональности в API (баллов за это не даётся).
5. Изображения. Нетрудно заметить, что изображение в предлагаемом API является легковесным объектом, а сами данные при этом расположены где-то ещё (как, например, это сделано в Vulkan). `AddImage` предполагает что Вы будете копировать входное содержимое изображения куда-то внутрь вашего API. Однако Вы можете изменить код в `main.cpp` так чтобы все загруженные изображения находились в различных векторах, вектора эти не удалялись и, таким образом, вам не нужно будет копировать данные изображений, а достаточно только скопировать ссылки на них, то есть саму структуру `Image2D`. Каковы последствия такого изменения?
 - (a) Преимущество в том, что ваш API, по сути, вообще не будет выделять внутри себя память (кроме Z-буфера, но и это можно исправить при желании). Это существенно для приложений повышенной надёжности и различных встраиваемых систем, где аллокация памяти строго статическая. Обратите внимание, что похожая концепция строгого контроля и не-выделения лишней памяти присутствует в Vulkan.
 - (b) Недостаток такого решения в том, что Вы не можете просто так втихую применить какие-то оптимизации. Например, хранить текстуру блоками, уменьшить битность текстуры или сделать обработку границ через добавление т.н. “юбки” вокруг текстуры (последнее является существенной оптимизацией в программных решениях при реализации текстурной выборки). А при копировании данных внутрь Вы можете это сделать.
 - (c) Именно по причине необходимости применения различных оптимизаций в Vulkan, например, мы имеем такой непростой порядок действий. Сначала Вы запраши-

ваете у API сколько памяти нужно для такой-то текстуры, потом выделяете память, и лишь затем создаёте объект текстуры который привязываете к выделенной памяти.

6. Рендер-пасс. Это понятие призвано сгруппировать определённую последовательность действий и/или функций отрисовки в **определенное изображение**.
 - (a) В простейшей реализации Вы очищаете нулями изображение в начале работы (*BeginRenderPass*) и синхронизируете работу вашего апи в конце (*EndRenderPass*) если, например, Вы используете многопоточность.
 - (b) Более сложный вариант может предполагать, что у Вас есть внутреннее представление буфера кадра, которой может храниться, например, тайлами 4x4. Такой формат повышает производительность блочных алгоритмов растеризации. В этом случае в *EndRenderPass* Вы также должны скопировать ваш внутренний буфер кадра в выходное изображение. Обратите внимание, что именно с подобными оптимизациями связаны связанные лэйауты текстур в Vulkan и тот факт, что рендер-пасс умеет их менять.
 - (c) Убедитесь что пользователь передаёт одно и то же изображение как в *BeginRenderPass*, так и в *EndRenderPass*. Гипотетически, нетрудно развить API таким образом, чтобы поддерживать одновременную отрисовку в различные изображения.

6.2 Стадии графического конвейера

Современный графический конвейер может быть представлен как набор параллельных процессов, где различные стадии конвейера связаны друг с другом по схеме производитель-потребитель (рис. 2). При этом предполагается, что данные передающиеся между стадиями, не выгружаются в оперативную память из сверх-оперативной, а передаются между потоками/выч.устройствами каким-то более эффективным способом.

Эффективная реализация такой схемы на современных вычислительных системах является нетривиальной задачей, поэтому многие программные решения просто распараллеливание отдельные стадии конвейера. Именно по этой причине в вашем задании параллельной реализации не требуется, и баллов за неё не даётся. Хотя Вы можете это сделать.

Итак, Вам необходимо реализовать следующую последовательность действий.

1. Вершинный шейдер;
 - (a) До выполнения вершинного шейдера необходимо перемножить матрицы *proj* и *worldView*, чтобы получить единственную матрицу, которая осуществляет преобразование из мирового пространства в т.н. NDC пространство (Normalized Device Coordinates – единичный куб от -1 до 1).

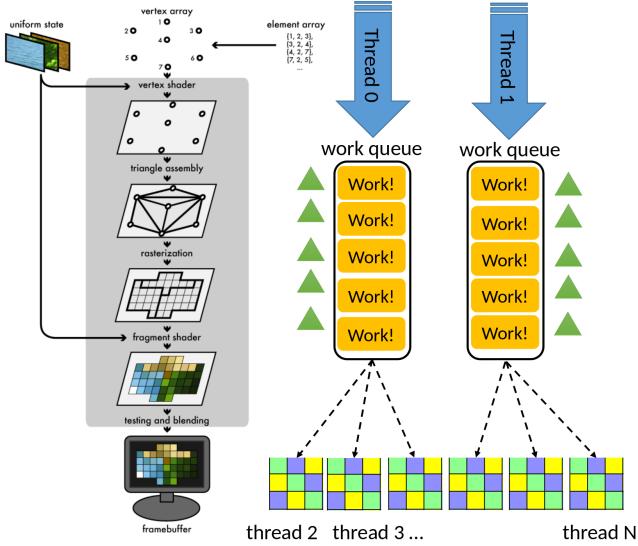


Рис. 2: Графический конвейер устроен по схеме производитель-потребитель, где каждая стадия принимает на вход порцию данных, обрабатывает её и передаёт ниже по конвейеру. Как правило, обработка фрагментов гораздо тяжелее геометрических стадий.

```
worldViewProj = projMatrix*worldView;
```

- (b) Необходимо умножить полученную матрицу worldViewProj на все вершины и, таким образом, преобразовать их.
- (c) Полученный вектор необходимо поделить на четвёртую координату “w” для того чтобы перейти от **Однородных Координат** [5] к декартовым.

```
vNDC[i] = worldViewProj*vpos4f[i];
```

```
vNDC[i] /= vNDC[i].w;
```

- (d) Итоговый вектор vNDC[i] необходимо перевести в пространство экрана:

```
x = vNDC[i].x*0.5 + 0.5 // [-1,1] -> [0,1]
```

```
y = vNDC[i].y*0.5 + 0.5 // [-1,1] -> [0,1]
```

```
x_screen = x*width - 0.5 // [0,1] -> [0,width]
```

```
y_screen = y*height - 0.5 // [0,1] -> [0,height]
```

- (e) В предлагаемом API матрицы хранятся по строкам в целях упрощения их восприятия. Однако внутри вашей реализации мы рекомендуем хранить матрицы по столбцам. Это позволяет компилятору оптимизировать умножение матрицы на точку с использованием векторных команд (используйте сайт godbolt [6] для проверки!).

2. Подготовка данных треугольников (TriangleSetUp или сборка примитивов). Эта стадия довольно проста. Её смысл заключается в том чтобы собрать все данные необходимые для растеризации треугольника в одну структуру, которая дальше передаётся по конвейеру. Поэтому всё что Вы должны сделать – прочитать три вершины треугольника по трём индексам из массива преобразованных вершин.
 - (a) Вы могли бы объединить стадию вершинного шейдера и стадию подготовки треугольника, однако в этом случае для вершин, входящих более чем в один треугольник вершинный шейдер выполнится более одного раза.
 - (b) Именно с этой проблемой связан т.н. T&L cache [7], суть которого в том, чтобы закэшировать некоторое количество преобразованных вершин чтобы далее сразу же передать их на стадию TriangleSetUp из вершинного шейдера, не выгружая в оперативную память.
3. Если Вы реализуете отбраковку обратных поверхностей по их порядку обхода, то именно на этой стадии.
4. Растеризация треугольников. Мы предлагаем Вам реализовать алгоритм растеризации на основе полу-пространства (в базе) и его блочную версию в дополнительной части. Подробности см. в следующем разделе.
5. В целях упрощения задания мы намеренно опускаем вопрос отсечения треугольников по ближней плоскости отсечения. В тестовых сценах геометрия расположена таким образом, что отсечение делать не требуется. Вам нужно лишь убедиться, что ограничивающий примитив прямоугольник не выходит за границу экрана.

6.3 Алгоритм растеризации на основе полу-пространства

Идея алгоритма растеризации на основе полу-пространства довольно проста: вы сканируете все пиксели в ограничивающем Ваш примитив прямоугольнике и проверяете, лежат центры пикселей внутри треугольника или снаружи (рис 3 b). В первом задании Вы уже сталкивались с функциями расстояния и решали подобную задачу для произвольных фигур. Так называемая edge-функция это и есть знаковое расстояние до ребра треугольника (формула 1).

$$E(A, B, P) = (P.x - A.x)(B.y - A.y) - (P.y - A.y)(B.x - A.x) \quad (1)$$

Здесь A и B две вершины ребра, а точка P – произвольная точка на плоскости изображения. Далее Вы просто реализуете вычисление этой функции для всех трёх рёбер треугольника $E_\alpha(x, y)$, $E_\beta(x, y)$, $E_\gamma(x, y)$: (формулы 2–4, рис 3 а)

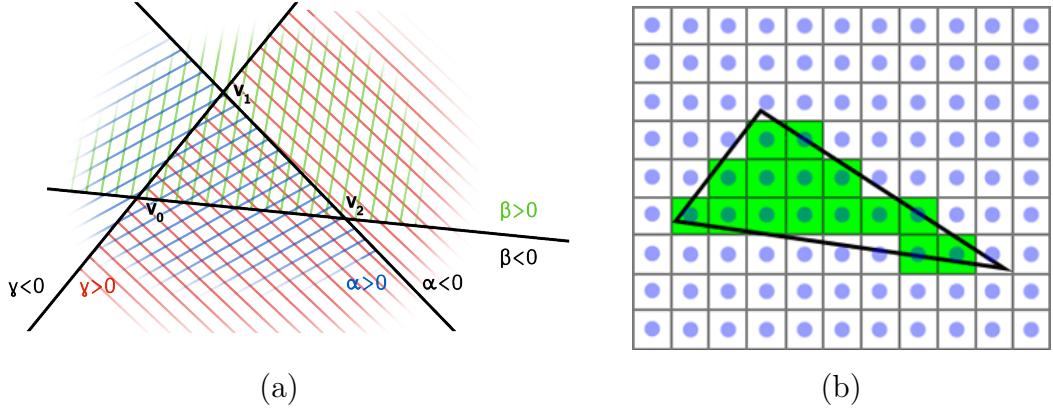


Рис. 3: Принцип работы растеризатора на основе полу-пространства (а); Стандартное поведение текста на перекрытие пикселя треугольником проверяет центр пикселя (б).

$$E_\alpha(x, y) = E(A, B, P) = (x - A.x)(B.y - A.y) - (y - A.y)(B.x - A.x); \quad (2)$$

$$E_\beta(x, y) = E(B, C, P) = (x - B.x)(C.y - B.y) - (y - B.y)(C.x - B.x); \quad (3)$$

$$E_\gamma(x, y) = E(C, A, P) = (x - C.x)(A.y - C.y) - (y - C.y)(A.x - C.x). \quad (4)$$

Однако для нас наиболее значимое свойство edge-функции в том, что она может быть **вычислена инкрементально** в процессе обхода ограничивающего прямоугольника. (См. алгоритм на рис. 4) [8]). Кроме того, барицентрические координаты $(u, v, 1 - u - v)$ могут быть напрямую вычислены из этой функции путём умножения значений edge-функции на обратную удвоенную площадь треугольника, которую также можно вычислить с помощью edge-функции (формулы 5–7).

$$u(P) = \frac{E(A, B, P)}{E(A, B, C)}; \quad (5)$$

$$v(P) = \frac{E(B, C, P)}{E(A, B, C)}; \quad (6)$$

$$w(P) = 1 - u(P) - v(P) = \frac{E(C, A, P)}{E(A, B, C)}. \quad (7)$$

```

1: for y in range minY .. maxY do
2:     Cx1 := Cy1;
3:     Cx2 := Cy2;
4:     Cx3 := Cy3;
5:     for x in range minX .. maxX do
6:         if Cx1 > 0 and Cx2 > 0 and Cx3 > 0 then
7:             u = Cx1*TriAreaInv;
8:             v = Cx2*TriAreaInv;
9:             framebuffer[x,y] := DrawPixel(u, v, 1-u-v);
10:            end if;
11:            Cx1 := Cx1 - Dy12;
12:            Cx2 := Cx2 - Dy23;
13:            Cx3 := Cx3 - Dy31;
14:        end for;
15:        Cy1 := Cy1 + Dx12;
16:        Cy2 := Cy2 + Dx23;
17:        Cy3 := Cy3 + Dx31;
18:    end for;

```

Рис. 4: Алгоритм растеризации на основе полу-пространства. Переменные $Cx*$ и $Cy*$ хранят значения edje-функции для строки и столбца изображения соответственно. Переменная $TriAreaInv = 1/E(A, B, C)$ это константа, которая равна обратной удвоенной площади треугольника. Тройка $(u, v, 1 - u - v)$ представляет собой значения барицентрических координат в центре пикселя. Переменные Dx^{**} и Dy^{**} представляют собой дельты, на которые изменяется edje-функция при сдвиге по x и y. Их вычисление описано в алгоритме на рис. 5.

```

1: x1 := tri.v3.y; // v3 or v1 dependeing on triangle vert order
2: x2 := tri.v2.y;
3: x3 := tri.v1.y; // v1 or v3 dependeing on triangle vert order
4: y1 := tri.v3.y; // v3 or v1 dependeing on triangle vert order
5: y2 := tri.v2.y;
6: y3 := tri.v1.y; // v1 or v3 dependeing on triangle vert order
7: Dx12 := x1 - x2;
8: Dx23 := x2 - x3;
9: Dx31 := x3 - x1;
10: Dy12 := y1 - y2;
11: Dy23 := y2 - y3;
12: Dy31 := y3 - y1;
13: Cy1 := (Dy12 * x1 - Dx12 * y1) + (Dx12 * minY - Dy12 * minX);
14: Cy2 := (Dy23 * x2 - Dx23 * y2) + (Dx23 * minY - Dy23 * minX);
15: Cy3 := (Dy31 * x3 - Dx31 * y3) + (Dx31 * minY - Dy31 * minX);

```

Рис. 5: Вычисление дельт и начальных значений для edge-функции. Переменные `minx` и `miny` содержат координаты минимума ограничивающего треугольник прямоугольника.

Вычисления в фиксированной точке. В вашем задании не требуется реализовывать алгоритм растеризации в фиксированной точке. Однако Вы должны учитывать, что любая программная или аппаратная реализация, претендующая на корректность будет реализовывать описанный выше алгоритм в фиксированной точке по очень простой причине: фиксированная точка даёт детерминированное поведение и стабильность. А кроме того, она существенно дешевле по аппаратной реализации и всё ещё быстрее на многих процессорах (в особенности тех которые не обладают аппаратной поддержкой плавающей точки).

6.4 Буфер глубины (z-буфер) и перспективная коррекция

Для того чтобы дальние объекты не перекрашивали при отрисовке ближние можно растеризовывать объекты в порядке от дальнего к ближнему (т.н. алгоритм художника). Однако такой алгоритм, во-первых, не слишком эффективен т.к. многие пиксели будут перекрашиваться по нескольку раз, а во-вторых, он не может обрабатывать случай проникновения одного объекта в другой (рис. 6).

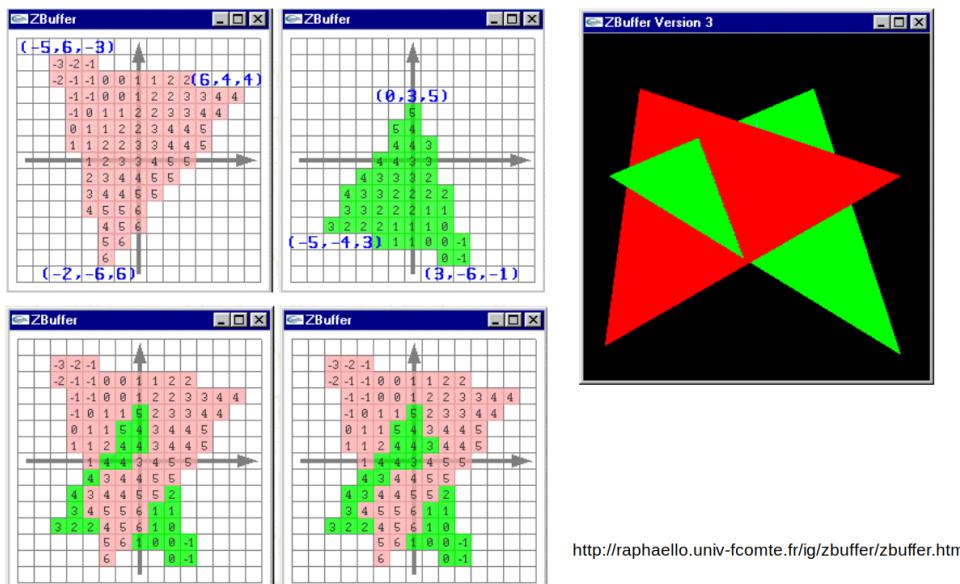


Рис. 6: Иллюстрация механизма работы алгоритма буфера глубины. независимо от порядки отрисовки треугольников мы получаем нужный результат, за исключением фрагментов (пикселей), который находятся на одинаковой глубине. Эта проблема называется Z-fighting.

Чтобы реализовать буфер глубины, Вам необходимо создать дополнительное изображение с глубиной с разрешением буфера кадра. Вы рисуете фрагмент и обновляете значения в буфере глубины только если тест глубины проходит – т.е. глубина текущего фрагмента меньше чем глубина, которая уже записана в z-буфер.

“ $1/z$ ” буфер и перспективная коррекция. Однако есть существенный нюанс, который мы до сих пор обходили стороной. Вспомните, что при переходе от однородных координат к декартовым мы делили координаты преобразованной вершины на w . Если посмотреть как обычно формируется матрица проекции, то можно увидеть что на самом деле в случае перспективной проекции деление на w будет соответствовать делению на z координату.

Преобразованная вершина будет состоять из следующих координат: $(x/w, y/w, z/w, 1/w)$. Если положить $w = z$, тогда $(x/z, y/z, 1, 1/z)$. То есть, после преобразования вершины в пространство NDC нам на самом деле нужны три числа: $(x/w, y/w, 1/w)$ которые условно соответствуют тройке $(x/z, y/z, 1/z)$.

Нюанс состоит в том, что поскольку в преобразовании координат из мирового пространства (или даже пространства камеры) в NDC присутствует операция деления, **нельзя просто так интерполировать значения вершинных атрибутов**, поскольку значение некоторого вершинного атрибута A при этом будет изменяться нелинейно при движении в плоскости изображения и обратном переходе в мировое пространство. А вот значение A/z (**вернее A/w**) **будет изменяться линейно!**

Поэтому в реализацию z -буфера и интерполяции вершинных атрибутов необходим внести следующие изменения:

1. В z -буфере вы храните не само значение z , а значение $1/z$ (вернее $1/w$);
2. На стадии TriangleSetUp для всех атрибутов (цвет, текстурные координаты др. если есть) вы умножаете их умножаете их значения на $(1/w)$:

$$tri.texCoord = texCoord * (1/w);$$
3. В каждом пикселе внутри треугольника Вы линейно интерполируете значение $(1/w)$ на основе тройки барицентрических координат;
4. Если тест глубины прошёл, Вы интерполируете при помощи тройки барицентрических координат значения атрибута поделённое на w ;
5. Перед непосредственным использованием атрибута (например выборкой из текстуры) в пикселе, его значение нужно привести в исходное пространство, поделив на проинтерполированное значение обратной глубины $(1/w)$.

6.5 Блочный алгоритм растеризации

Описанный алгоритм можно очевидным образом улучшить, если проверять не отдельные пиксели, а углы блоков, скажем 4×4 или 8×8 пикселей. За деталями мы рекомендуем обращаться к работе [9].

Список литературы

- [1] Шаблон для выполнения задания: https://github.com/FROL256/sw_gapi_task
- [2] Bommes, David and Lévy, Bruno and Pietroni, Nico and Puppo, Enrico and Silva, Claudio and Tarini, Marco and Zorin, Denis. Quad-Mesh Generation and Processing: A Survey. Computer Graphics Forum (2013). 32. 10.1111/cgf.12014.
- [3] Подготовленные для задания меши из квадов.
URL = <https://disk.yandex.ru/d/fX-95hx7Z8nDJA>
- [4] Rasterization: a Practical Implementation, Scratchapixel 3.0. A free educational site that progressively introduces you to the world of computer graphics. <https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.html>
- [5] Алексей Игнатенко. Однородные координаты. Исследовать в интернете.
- [6] Compiler Explorer: <https://godbolt.org/>
- [7] Cem Cebenoyan and Matthias Wloka. Optimizing the Graphics Pipeline. https://www.nvidia.pl/docs/IO/8228/GDC2003_PipelinePerformance.pdf
- [8] Juan Pineda. 1988. A parallel algorithm for polygon rasterization. SIGGRAPH Comput. Graph. 22, 4 (Aug. 1988), 17–20. <https://doi.org/10.1145/378456.378457>
- [9] Péter Mileff, Károly Nehéz, Judit Dudra. Accelerated Half-Space Triangle Rasterization. (2015) http://acta.uni-obuda.hu/Mileff_Nehez_Dudra_63.pdf