УДК 004.21

АЛГОРИТМ ВИЗУАЛИЗАЦИИ МНОЖЕСТВА ГРАФИЧЕСКИХ ПРИМИТИВОВ ДЛЯ ГРАФИЧЕСКОГО ПРОЦЕССОРА С ИСПОЛЬЗОВАНИЕМ РАЗЛИЧНЫХ СТИЛЕЙ ОТОБРАЖЕНИЯ И КОНСТРУКТИВНОЙ ГЕОМЕТРИИ В ДВУХ ИЗМЕРЕНИЯХ

Мирзоян Д.И.

МИРЭА – Российский технологический университет, 119454, г. Москва, пр-т Вернадского, 78, e-mail: d.i.mirzoyan@gmail.com

Используемые при разработке средств автоматизированного проектирования электронных устройств алгоритмы визуализации не имеют достаточной производительности для работы с современными задачами большого объёма, такими как разработка чиплетов. Целью работы является разработка алгоритма визуализации на базе графического процессора, имеющего аналогичные функциональные возможности и достаточную для решения сложных современных и перспективных задач производительность. Алгоритм основывается на идее обработки геометрических примитивов в растровом виде на графическом процессоре, вычисление результирующей геометрии на центральном процессоре не производится. От алгоритма требуется формирование контура результирующей геометрии, параметрических заливок и работа с полупрозрачностью. Разработанный алгоритм успешно справляется с задачами объёмом в 1,6 млн. элементов и имеет некоторый резерв производительности и возможность дальнейшего улучшения.

Ключевые слова: графический процессор, визуализация, шейдеры, двумерная графика, САПР, EDA CAD, 2D CAD, электронные устройства

GPU-BASED ALGORITHM OF TWO-DIMENSIONAL CONSTRUCTIVE GEOMETRY

Mirzoyan D.I.

MIREA - Russian Technological University, 119454, Russia, Moscow, Vernadsky Avenue, 78, e-mail: d.i.mirzoyan@gmail.com

The visualization algorithms used in the development of electronic design automation computer-aided design software do not have sufficient performance to work with modern large-scale tasks, such as the development of chiplets. The aim of the work is to develop a visualization algorithm based on a graphics processor that has similar functionality and sufficient performance to solve complex modern and promising tasks. The algorithm is based on the idea of processing geometric primitives in raster form on the GPU, while resulting geometry in vector form is not being calculated on the CPU. The algorithm is required to form the outline of the resulting geometry, parametric fills and work with transparency. The developed algorithm successfully copes with tasks with a volume of 1.6 million elements and has some performance reserve and the possibility of further improvement.

Keywords: GPU, visualization, shaders, 2D graphics, CAD, EDA CAD, 2D CAD, electronic devices

Введение

В исторический период перехода системных шин персональных компьютеров со стандарта ISA на стандарт PCI существовал класс устройств, известных как «акселераторы Windows». Это были видеоадаптеры со специальными механизмами ускорения отрисовки двумерной графики. Их появление было связано с ростом разрешения дисплеев и недостаточной производительности шины ISA для обеспечения приемлемой для работы пользователя кадровой частоты графического режима. После перехода на шину PCI, имевшую требуемую производительность, данный класс устройств потерял актуальность и более не производился. Следующий этап возникновения специальных интерфейсов в графических адаптерах уже был связан с распространением трехмерной графики. Расчет трехмерных сцен является весьма вычислительно сложной задачей, и современные процессоры общего назначения обычно не имеют достаточно вычислительной мощности для обеспечения

комфортной для пользователя производительности (кадровой частоты). Для обработки задач трехмерной графики были разработаны и сегодня повсеместно используются специализированные графические процессоры.

Несмотря на первичную специализацию под трехмерную графику, графические процессоры могут успешно решать и задачи двумерной графики, такие как попиксельное отсечение, альфа-смешивание и т.д. Эта возможность успешно используется современными операционными системами для формирования изображения пользовательского интерфейса. Основной причиной является бОльшая энергетическая эффективность данных операций в исполнении графического процессора, формирование основных частей пользовательского интерфейса выполняется силами центрального процессора.

БОльшая часть двумерных САПР используют для отображения графики интерфейсы библиотеки пользовательского интерфейса (GTK+, Qt) или операционной системы (GDI, GDI+). В обоих случаях основные задачи формирования изображений решаются с использованием центрального процессора, графический процессор используется ограниченно для отдельных операций.

При этом объем задач, решаемых подобными САПР, постоянно растет, и в некоторых случаях (вроде разработки подложек чиплетов) использование для отображения центрального процессора не способно обеспечить достаточный уровень производительности [1]. При переносе задачи отображения на графический процессор возникают дополнительные проблемы, как архитектурные (взаимодействие с графическим процессором асинхронно, в отличие от синхронной отрисовки центральным процессором, набор примитивов графического процессора отличается от набора примитивов большинства библиотек пользовательского интерфейса), так и алгоритмические (графический процессор не умеет осуществлять зональную заливку, непосредственно отображать текст и т.п.). Перенос задач отображения на графический процессор позволяет освободить центральный процессор для других прикладных задач, не настолько требовательных к времени отклика, например, численному моделированию тепловых и/или электромагнитных процессов. Типичный объём этих задач делает невозможным их решение в реальном времени с приемлемой для практического применения точностью [2].

При переносе процесса отображения на графический процессор конкретно для класса программ двумерных САПР актуальными являются следующие проблемы:

1) Формирование контуров примитивов
2) Формирование процедурных заливок примитивов
3) Отображение текста
4) Отображение служебной графики — направляющих линий, сеток, курсоров и т.п.

Заливка текстурой при этом проблемой не является — использование стандартных для трехмерной графики механизмов текстурирования успешно решает данную задачу.

Пункты 1 и 2 являются важными для визуального различения примитивов, обозначающих различные объекты предметной области.

В большинстве библиотек пользовательского интерфейса и графических библиотек двумерной графики используется примерно одинаковый набор примитивов и функциональных возможностей.

Основные используемые примитивы (рис. 1):

1) отрезок прямой (на рисунке не показан)
2) эллипс
3) прямоугольник
4) дуга/сектор окружности/эллипса
5) многоугольник
6) кривая Безье
7) ломаная (полилиния, polyline)
8) окружность
9) прямоугольник

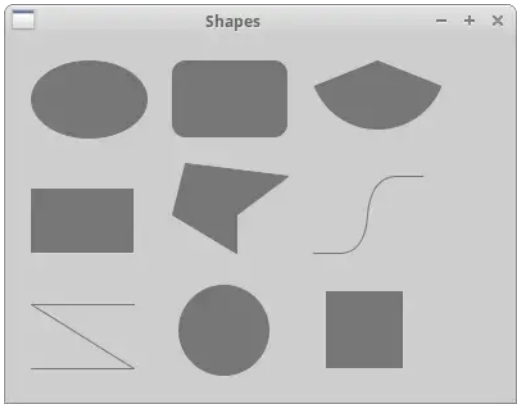

Рисунок 1 — Основные примитивы Windows GDI

При этом некоторые примитивы могут иметь ограничения, обусловленные программным интерфейсом и реализацией. Например, стороны прямоугольника строго параллельны краям дисплея, в таком случае для определения прямоугольника достаточно 2 точек. При необходимости отобразить прямоугольник с

непараллельными краям экрана сторонами используется примитив «многоугольник». Аналогичные ограничения есть для эллипсов и дуг.

Также для отображения используются абстракции:

1) «перо» — задает параметры отображения контуров/линий: стиль, цвет, толщину
2) «кисть» — задает параметры отображения заливок сплошных фигур: стиль, цвет, текстуру
3) «шрифт» — задает параметры отображения текста: гарнитуру, размер, атрибуты (подчеркнутый, курсивный, полужирный).

Разнообразие примитивов используется для графического представления элементов предметной области, а перья, кисти и шрифты — для легкого визуального различения отдельных объектов. Следует отметить, что в пользовательском интерфейсе обычных программ используется ограниченный набор комбинаций примитивов и стилей, обычно выбранных в соответствии с рекомендациями для ПИ соответствующей ОС. Программное обеспечение, работающее со сложными задачами, для представления объектов использует более широкий набор стилей отображения. Примеры стандартных стилей «кистей» приведены на рис.2.

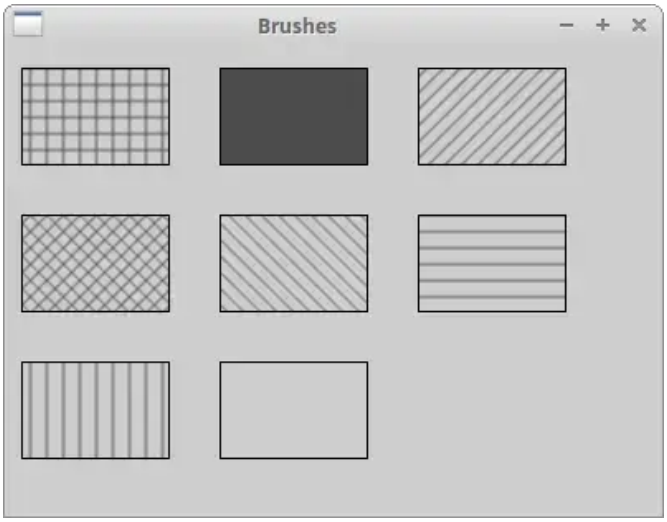

Рисунок 2 — Предопределенные заливки Windows GDI

В зависимости от предметной области и устоявшихся для данной области, для данного поставщика ПО или даже для конкретного продукта для различения элементов отображаемой сцены могут использоваться как различные варианты стилей заливок / стилей контуров / толщин контуров, так и цветовая дифференциация. Существуют продукты, поддерживающие оба механизма, в таком случае выбор визуального представления групп элементов остаётся за пользователем.

Поскольку для растеризации в данном случае используются вычислительные ресурсы центрального процессора, визуализация задач большого объёма сталкивается с недостатком вычислительных мощностей. Стандартные же механизмы трёхмерной графики не имеют такого широкого выбора примитивов и стилей отображения, хотя и имеют преимущество в простоте расчётов цвета и (полу)прозрачности.

Постановка задачи

Наблюдается востребованность возможности отображения сложных графических сцен в некоторых предметных областях с использованием более мощных механизмов, чем растеризация силами центрального процессора. Современные реализации описанных механизмов отображения ограниченно используют мощности графического процессора, например, для операций попиксельного отсечения и альфа-смешивания. Перспективным представляется перенос всего процесса отображения на графический процессор, с исключением центрального из растеризации примитивов и освобождением его ресурсов для других прикладных задач.

До определенного момента развития решение данной проблемы с использованием графического процессора было затруднено, если не невозможно — набор функциональных блоков и их взаимосвязей был сильно ограничен и подчинён основной задаче — отображению трёхмерной графики. Ситуация изменилась сперва с появлением частичной программируемости отдельных узлов (вершинных и фрагментных процессоров), затем с переходом к унифицированной архитектуре графического процессора, в котором большая часть задач, ранее разбитых на отдельные аппаратные блоки, выполняется программно на массиве унифицированных ядер. Поскольку вычисление атрибутов пикселя (цвет, прозрачность) на графических процессорах унифицированной архитектуры выполняется полностью программируемо, и не зависит линейно от исходных данных, стало возможно решить задачу переноса стилей отображения и набора примитивов библиотек пользовательского интерфейса на графический процессор методом «от обратного». Т.е. не растеризовать определенную стилистическую часть примитива (заливку, контур), а определять, принадлежит ли пиксель контуру/заливке/вырезу или иному стилистическому элементу.

Данную задачу можно архитектурно разделить на две подзадачи:

1) Составление набора примитивов и механизмов их комбинации и преобразований, с одной стороны, легко отображаемых графическим процессором, с другой стороны, позволяющим графически представить все элементы предметной области

2) Реализацию алгоритма, позволяющего использовать механизмы визуальной стилизации, привычные для библиотек пользовательского интерфейса

Для решения задач визуализации сложных графически сцен предметной области широко применяются алгоритмы с использованием графического процессора. Чаще всего ускорение визуализации графическим процессором используется в задачах трёхмерного пространства [3, 4]. Но, например, в САПР электронных устройств также часто используется визуализация с помощью графического процессора, хотя основные задачи данных САПР являются двумерными [5]. Перенос визуализации на графический процессор позволяет освободить мощности центрального процессора для решения других задач, таких как, например, проверка правил проектирования в реальном времени. Перенос на графический процессор ещё и задачи геометрических вычислений позволит одновременно выделить больше мощностей центрального процессора для решения второстепенных задач и существенно увеличить количество обрабатываемых примитивных визуальных элементов предметной области.

Решение подзадачи 1 состоит в формировании набора примитивов, позволяющего эффективно их обрабатывать с использованием графического процессора и достаточного для формирования из них визуального представления примитивов, обычно использующихся для объектов предметной области. Соответствующее алгоритмическое обеспечение представляет собой алгоритм комбинирования примитивов для формирования единого визуального представления — базовый набор геометрических операций, аналогичный конструктивной геометрии в двух измерениях.

Полный набор геометрических операций с отдельными примитивами может быть представлен в виде дерева (рис. 3а). Количество листьев этого дерева соответствует количеству примитивных визуальных элементов предметной области, и в рассматриваемых условиях может достигать нескольких миллионов штук. Настолько разветвлённые структуры данных неэффективны при обработке графическим процессором [6], поэтому необходимо архитектурное и/или алгоритмическое решение, снижающее сложность обрабатываемых структур данных. Высокая параллельность графического процессора позволяет решить эту проблему с другой стороны. Вместо вычисления сложной итоговой фигуры и её растеризации можно определить принадлежность каждого визуализируемого пикселя к тому или иному примитиву или группе примитивов. В таком случае можно объединить множество однотипных операций в группу, и рассчитывать их параллельно. При этом существенно уменьшится количество узлов дерева операций (рис. 1б). Такая группировка примитивов является наиболее эффективной в силу специфики САПР электронных устройств. В большинстве случаев все операции могут быть сведены к дереву глубиной 4 уровня с не более чем 3 дочерними узлами в каждом узле 2-го уровня.

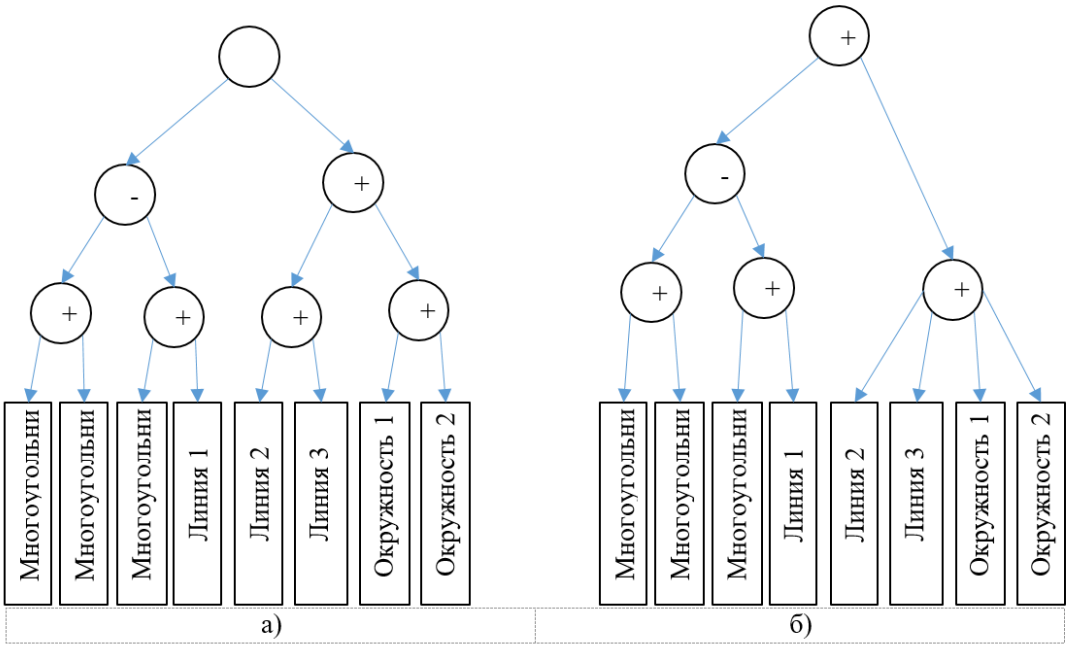

Рисунок 3 — Деревья геометрических операции с примитивами, возникающие в САПР электронных устройств

Следует заметить, что дерево, изображённое на рисунке 1а, является строго двоичным, дерево же с группировкой операций по типу, изображённое на рисунке 3б, является деревом общего вида, без ограничений на количество элементов. При этом двоичное дерево может иметь любую высоту, высота же дерева на рис 3б ограничена 4 уровнями. Это позволяет использовать эффективные в случае графического процессора структуры данных последовательного доступа (массивы элементов) вместо структур данных произвольного доступа (деревья) [7]. Применительно к структурам, характерным для графического процессора, можно сказать, что для каждой операции 3-го уровня дерева, непосредственно работающей с элементами (примитивами), выделяется отдельный буфер вершин. Из-за специфики архитектуры графического процессора для обеспечения высокой производительности все листья данного дерева находятся именно на уровне 4. В случае необходимости пустые уровни дерева заполняются тривиальными операциями.

После вычисления набора координат пикселей, принадлежащих определенному объекту предметной области, необходимо рассчитать их визуальную составляющую (цвет), что приводит к необходимости решения подзадачи №2. Поскольку в унифицированной архитектуре графического процессора наборы данных, передаваемых программам не являются фиксированными и имеют достаточно мягкие ограничения по объёму, мы можем передать всю необходимую для отображения контуров и заливок информацию прямо в программу графического процессора (шейдер). Основным элементом данных в таком случае является собственно стиль заливки — некоторое число, обозначающее алгоритм, по которому будет строится процедурная текстура (рис. 4). Препятствием для наивной реализации является факт, что графический процессор не оперирует пикселями, а особенно их координатами напрямую — фрагментный шейдер, вычисляющий цвет конкретного пикселя не знает автоматически его координат, как это происходит в алгоритмах растеризации для центрального процессора, ему доступен только набор данных, обозначенный разработчиком при создании шейдерной программы. Если рассматривать вопрос собственных координат пикселя, с точки зрения графического процессора — это всегда вещественное число в диапазоне [-1; 1] по x и y. Необходимо перевести собственные координаты пикселя в экранное пространство, поскольку все элементы стилей имеют именно пиксель как базовую единицу.

В приложениях трёхмерной графики для преобразования координат примитивов используется набор из трёх матриц:

1) матрица модели, преобразующая координаты из собственной системы счисления модели в систему координат мира
2) матрица вида, преобразующая координаты из системы координат мира в систему координат камеры
3) матрица проекции, преобразующая систему координат камеры к кубу отсечения

В современных приложениях матрицы 1 и 2 объединены в единую матрицу модели-вида (modelview).

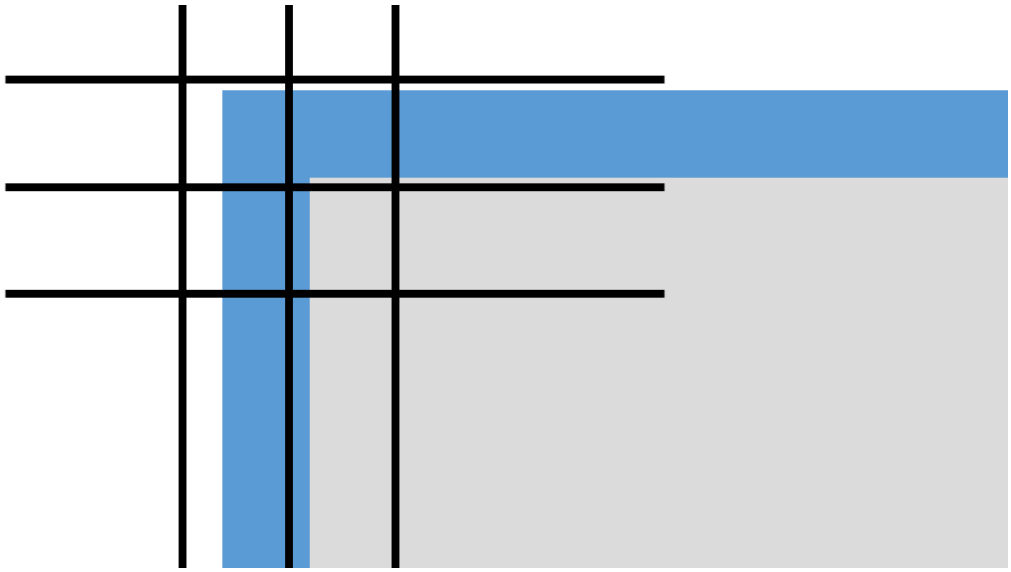

Рисунок 4 — Расчёт стилей примитива при смещении относительно сетки пикселей

Подход к построению структуры данных, показанный на рисунке 3б в целом соответствует естественной структуре предметной области САПР электронных устройств. Предметная область САПР электронных устройств состоит из отдельных слоёв. При визуализации совмещение слоёв реализует операция геометрического сложения на 1-м уровне дерева. Каждый слой разделён на «позитивные» (области металлизации) и «негативные» (области травления) подслои, причём суммарное количество этих подслоёв в слое обычно не превышает 3. Отдельные

примитивы являются частью подслоя.

На графическом процессоре процесс визуализации данной структуры данных происходит в 4 этапа (стадии):

1) визуализация каждого подслоя в соответствии с его типом («позитивный» или «негативный»);
2) сведение отдельных изображений подслоёв в изображение слоя в соответствии с типом подслоя: «негативные» подслои вычитаются из «позитивных», «позитивные» складываются;
3) сведение слоёв в соответствии с настройками каждого слоя (цвет, заливка, порядок в стопке, видимость и т.п.) в единое изображение;
4) постобработка изображения (наложение сетки, чертёжных линий и пр.) и вывод результата на монитор.

Для формирования промежуточных изображений на этапах 1, 2, 3 осуществляется рендеринг в текстуру, для хранения результатов применяются текстурные массивы на необходимое количество элементов.

Конкретный набор примитивов зависит от реализации САПР электронных устройств. В случае использования программной прорисовки, обычно используется набор примитивов, предоставленный графической библиотекой. В случае реализации переносимой архитектуры, как в [8], набор примитивов должен быть достаточно прост для реализации на графическом процессоре и достаточно сложен для обеспечения полноты функциональных возможностей. При реализации отрисовки исключительно на графическом процессоре набор примитивов сходен с предыдущим случаем (или незначительно меньше), поскольку предоставляемых аппаратно примитивов, очевидно, недостаточно для решения задачи (в частности, полностью отсутствуют окружности и дуги). Данная работа является частью реализации архитектурного подхода, описанного в [8], и в данном случае набор примитивов включает в себя:

1) окружность;
2) кольцо;
3) дуга окружности;
4) плоскоовал;
5) треугольник;
6) четырёхугольник;
7) произвольный многоугольник;
8) треугольник с закруглёнными углами;
9) четырёхугольник с закруглёнными углами;
10) произвольный многоугольник с закруглёнными углами;
11) чертёжный отрезок;
12) чертёжная дуга.

Отдельно следует отметить суть примитивов №11 и №12 — они геометрически эквивалентны примитивам №4 и №3 соответственно, но толщина задаётся в видимых пикселях и не зависит от масштаба отображения. Данные примитивы используются для реализации служебных отметок (для логического выделения областей). Выделение треугольников и четырёхугольников из произвольных многоугольников обусловлено существенно более простой обработкой их вершин (триангуляцией).

Все примитивы преобразуются в наборы геометрических параметров (вершин) и передаются в память графического процессора, где и хранятся. При изменении параметров сцены, не затрагивающих геометрию элементов, перезаписи данных в памяти графического процессора не происходит. Определение принадлежности визуализируемых точек к примитиву в нетривиальных случаях (окружность, кольцо, плоскоовал, многоугольники с закруглёнными углами) осуществляется в процессе исполнения шейдерной программы на графическом процессоре. Всего используется набор из 4 шейдерных программ, таких как:

1) визуализация «позитивного» подслоя (стадия 1);
2) визуализация «негативного» подслоя (стадия 1);
3) сведение подслоёв в слой (стадия 2);
4) сведение слоёв в изображение и постобработка (стадия 3 и стадия 4).

Шейдерные программы №1 и №2 формируют в своих выходных буферах не изображение, а наборы параметров для каждого пикселя. Поэтому их непосредственная визуализация не имеет смысла, и промежуточные результаты работы для данной стадии не приведены.

Шейдерная программа №1 выводит в текстуру параметры пикселей примитивов. Все входящие примитивы при этом объединяются геометрически операцией «или», т.е. осуществляется геометрическое сложение.

Шейдерная программа №2 действует полностью аналогичным образом, за исключением обработки контуров (см. далее).

Шейдерная программа №3 вычисляет цвет пикселя итогового изображения на основании параметров, записанных во входном текстурном массиве. При это фактически осуществляется и операция геометрического

«вычитания» одно подслоя из другого. Результат работы в виде графического изображения с каналом прозрачности записывается в элемент текстурного массива, являющийся входными данными для этапа №4.

Шейдерная программа №4 осуществляет сведение слоёв по порядку (они записываются в элементы входного текстурного массива в порядке следования) с учётом прозрачности отдельных элементов и настроек прозрачности слоя. Далее на изображение накладывается настраиваемая глобальными параметрами шейдерной программы координатная сетка.

Итоговый результат визуализации приведён на рис.5. На рисунке 5 виден сплошной зелёный многоугольник заливки (полигон) на верхнем слое, являющийся часть подслоя 1 верхнего слоя. Подслой 1 верхнего слоя является «позитивным». В полигоне сделаны вырезы вокруг дорожек, находящиеся в подслое 2. Подслой 2 в таком случае является «негативным». В подслое 3 находятся сами дорожки и металлизация вокруг переходных отверстий. Подслой 3 также является «позитивным». В вырезах верхнего слоя виден жёлтый многоугольник следующего слоя, а в вырезах вокруг переходных отверстий — и последующие слои. Реальный проект соответствует структуре слоёв и геометрических операции с примитивами, рассмотренными выше.

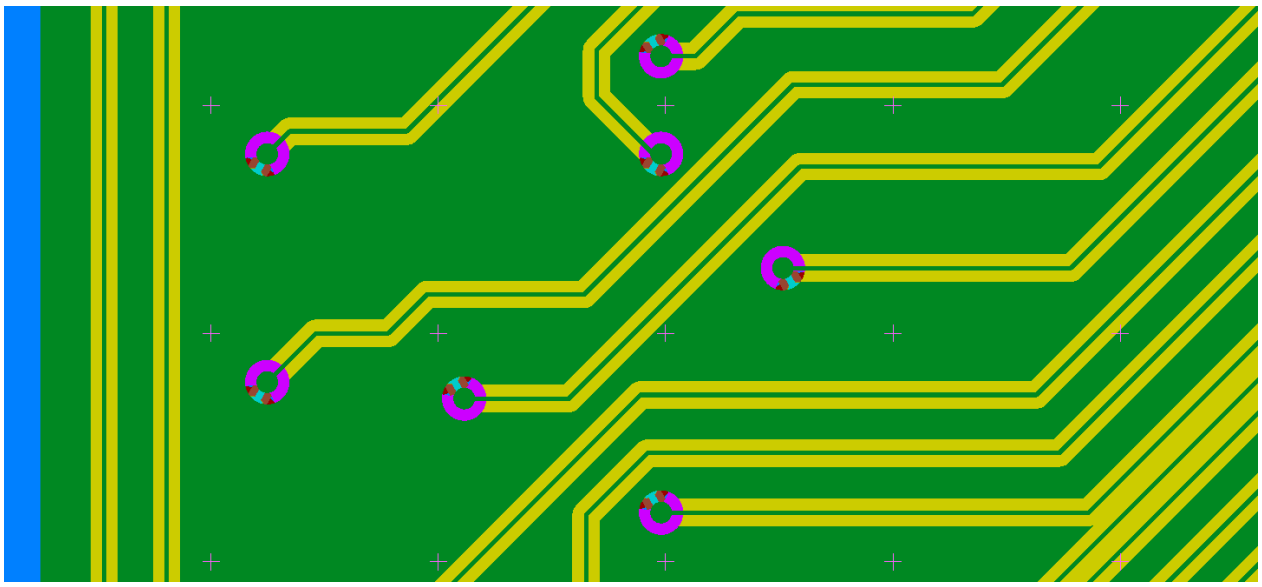

Рисунок 5 — Пример визуализации части рабочего проекта

В некоторых САПР для повышения наглядности отображения и обеспечения возможности визуального разделения большого количества типов элементов применяются дополнительные параметры визуализации, такие как толщина контура и тип заливки [9]. Данный подход позволяет визуально выделять элементы в случаях, когда выделение цветом не даёт нужного эффекта — человек может воспринимать большое количество цветов, но быстро выделять цвета на сцене с большим количеством элементов может быть некомфортно. Кроме того, различение цветов зависит от индивидульного цветовосприятия и качества и функциональных возможностей отображающей аппаратуры. Эти факторы делают возможность использования различных типов заливок достаточно важной. Пример визуализации элемента с заливкой приведён на рис. 6.

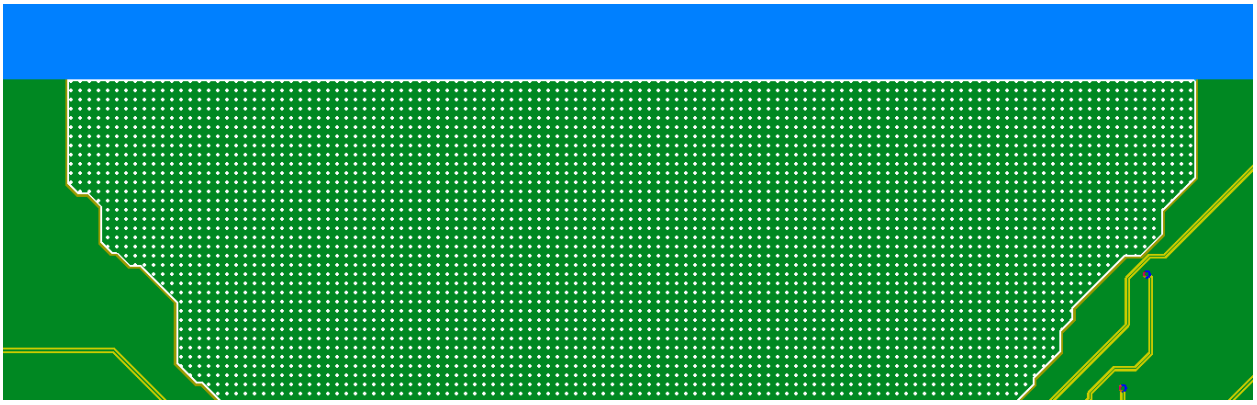

Рисунок 6 — Визуализация элемента с контуром и заливкой при выполнении операции геометрического «вычитания».

Для формирования изображения контура и заливки необходимо в шейдерных программах №1 и №2 математически определить, относится ли пиксель к контуру, заливке, или иному стилистическому элементу.

Напрямую определить экранные координаты пикселя аппаратная модель работы графического процессора и программная библиотека разработки шейдерных программ не позволяют, но данная проблема может быть решена косвенно. Поскольку полный набор матричных преобразований в конечном итоге даёт координаты пикселя в экранном пространстве (кадровом буфере), то возможно повторить в коде шейдерной программы все необходимые расчёты, основывающиеся на некоторых известных приложению параметрах.

Список необходимых параметров включает

1) смещение области просмотра относительно начала координат
2) вращение области просмотра
3) размерность сторон области отображения
4) коэффициент масштабирования области просмотра

Сочетание перечисленных параметров позволяет вести расчёты в пиксельной системе координат. Все расчёты при этом ведутся в вещественных числах, что вызывает неточность совпадения элементов и рассчитанных координат (рис. 7), в случаях, например, смещения на нецелое количество пикселей или отображения объектов, имеющих дробный размер при данном коэффициенте масштабирования.

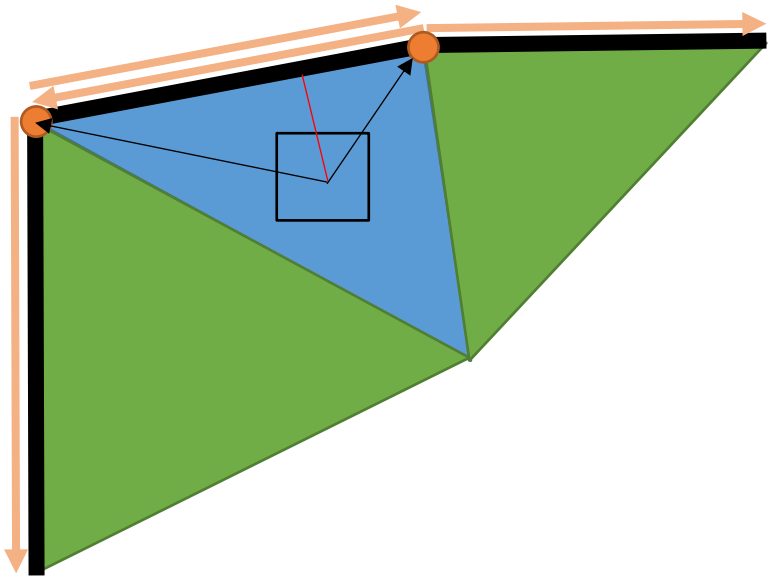

Рисунок 7 — расчёт принадлежности пикселя к стилистическому элементу

Решением данной проблемы является подход к пикселю не как к математической точке, что имеет место в целочисленных алгоритмах растеризации, а как к протяженному геометрически объекту (квадрату). В таком случае строгого совпадения рассчитанных координат с решеткой не требуется. У данного подхода тоже есть недостаток — поскольку мы не знаем, выполнялось ли отображение данного элемента в данном месте или нет, при определенных ситуациях, вызванных особенностями расчётов и округлений вещественных чисел одинарной точности, происходит дублирование (когда 2 пикселя «попадают» в элемент с шириной в 1) либо исчезновение (когда элемент попадает «между» пикселями). Данная проблема вызвана именно представлением вещественных чисел и решается введением добавочной константы.

Шейдерной программе не может быть доступен полный набор данных о примитиве в силу ограничений архитектуры графического процессора. Набор передаваемых шейдерной программе данных может быть велик, но не бесконечен. Также передача большого объёма параметров связана с существенной потерей производительности. Следует ограничить объём передаваемых данных минимально-необходимым набором. Для выявления необходимого набора данных рассмотрим процесс расчёта с точки зрения отдельного пикселя (рис. 7).

Шейдерной программе неизвестны и не могут быть известны цвета и принадлежность соседних пикселей. С этой стороны определить край примитива невозможно. Использовать параметры самого примитива графического процессора тоже можно лишь ограниченно — некоторые задачи требуют манипуляций примитивами с появлением несоответствий между требуемыми визуальными границами и математическими представлениями примитивов. Поскольку базовым примитивом графического процессора является треугольник, рассмотрим

визуализацию сцены, составленной из треугольников.

Треугольник имеет три стороны и три вершины. Соответственно, для расчётов контуров достаточно указать в экранной системе координат три вершины и три отрезка. Мало какие предметные области могут быть представлены одними треугольниками, поэтому из треугольников собираются более сложные примитивы (многоугольники). Разбиение многоугольника на треугольники может выполнятся различными алгоритмами, но любой из них может (в зависимости от многоугольника) порождать треугольники, состыкованные сторонами друг с другом — в таком случае сторона является внутренней частью примитива и не должна выделяться контуром. Вершины треугольников при этом соответствуют вершинам оригинального примитива.

Рассмотрев все возможные сценарии отношений треугольника в составе более сложного многоугольника, необходимый объем данных был определен как 3 вершины, соответствующие вершинам треугольника и 3 пары (6 сторон) отрезков, соответствующие возможным граням многоугольника.

Для определения принадлежности пикселя контуру, заливке, вырезу или другому стилистическому элементу, вычисляются расстояния до 3 точек и 6 сторон. В случае, если пиксель попадает в контур хоть в одном случае, он считается принадлежащим контуру, с применением соответствующего цвета. Если пиксель попадает в вырез, его обработка завершается без сохранения каких-либо результатов. В оставшемся случае пиксель считается принадлежащим заливке.

На рисунке 7 приведён частный случай подобного расчёта. Пиксель (отображён черной квадратной рамкой) имеет определённые координаты, в данном случае считающиеся от центра пикселя. Точки контура примитива (оранжевые кружки) имеют по две прилежащие стороны (оранжевые стрелки). Путём базовых геометрических расчётов определяется расстояние центра пикселя до стороны (красный отрезок). На основании этого расстояния и преданных в качестве параметров описаний стилистических элементов выносится решение о принадлежности пикселя контуру, заливке или иному стилистическому элементу.

Данный механизм расчёта позволяет отрисовывать контур в трёх вариациях — полностью находящийся внутри примитива, центр контура совпадает с границей примитива и контур полностью за границей примитива. Также возможна отрисовка законтурных эффектов (тень, свечение), при реализации соответствующего расширения примитива.

Некоторые рассматриваемые примитивы (эллипсы, окружности, дуги, кривые Безъе) затруднительно набрать из отдельных треугольников. Их можно определить параметрически, определив принадлежность пикселя примитиву на этапе вычисления цвета, не создавая сложный набор аппроксимирующих треугольников. Также такой подход даёт более гладкий контур полученной фигуры.

Для отображения окружностей они представляются в виде описывающего прямоугольника, состоящего из двух треугольников, а в шейдерную программу дополнительно передаются координаты центра окружности и её радиус. Для представления эллипса вводится дополнительный множитель по одной из осей.

Описанный алгоритм позволяет отображать примитивы с заданной заливкой и контуром заданной толщины, используя вычислительный ресурс преимущественно графического процессора. Центральный процессор при этом отвечает только за формирование наборов параметров примитивов, причём этот набор инвариантен относительно области просмотра. Все изменения области просмотра передаются через небольшой набор коэффициентов, без изменения основного набора данных. Сложности вызывает расчёт стилей контуров (рис 8). Требуется введение дополнительного параметра (одномерная линейная координата по контуру) с расчётом на стороне центрального процессора. Учитывая распространение дисплеев высокого разрешения, на которых тонкие (1-2 пикселя) линии со стилями, отличными от «сплошного», выглядят недостаточно различимо, данный элемент при реализации был опущен. Второй проблемой является отображение кривых Безъе, поскольку для них отсутствует достаточно быстрый в реализации алгоритм определения принадлежности точки кривой.

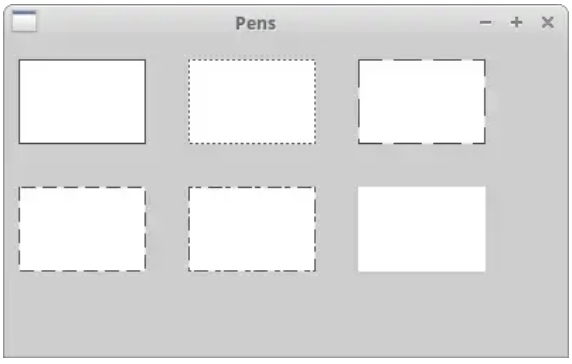

Рисунок 8 — Предопределенные стили перьев (контуров) Windows GDI

Третьей проблемой в данном случае является операция «вычитания» элементов, не формирующая очевидным образом контура. В традиционных реализациях с использованием центрального процессора контуры всех фигур получаются очевидным образом в результате расчёта геометрических операций. Собственно, контур и является результатом подобных расчётов. В случае графического процессора обработка идёт на уровне точек (пикселей изображения), и в описанном выше алгоритме нет очевидного способа определения принадлежности точки контуру результирующей фигуры при выполнении операции вычитания. Возможно при проверке каждой точки выполнять поиск границы в её окрестности с радиусом равным толщине контура, но это приведёт к существенному увеличению количества операций с памятью и, как следствие, к критической потере производительности. Для решения данной проблемы предлагается в отдельном буфере отмечать возможную принадлежность точки контуру. При этом необходимо учитывать приоритет областей с заливкой / пустых областей в «позитивных» и «негативных» областях соответственно. В графических фреймворках предусмотрено несколько типов буферов: буфер цвета, буфер глубины, буфер аккумулятора и буфер трафарета. Последние два типа буферов обладают ограниченной поддержкой в современных поколениях графических процессоров и непригодны для данной цели. Буфер цвета потенциально возможно использовать, но он не даёт возможности автоматического учёта приоритета областей контура и заливки. Буфер глубины (z-буфер) обычно используется для отсечения невидимых поверхностей и монопольно занят выполнением данной функции. Однако, в случае работы в двух измерениях, с явным разбиением на слои, использование буфера глубины по его основному назначению не требуется, и становится возможным применять его для решения вспомогательных задач. Воспользуемся z-буфером для устранения проблемы контуров при выполнении геометрических операции «сложения» и «вычитания». При этом нужно учесть возможность наличия фигур с различной толщиной контура. Определим для каждого слоя максимальную толщину контура и передадим этот параметр в шейдерную программу. Вокруг геометрических фигур для автоматического учёта контуров формируется область с градиентным изменением виртуального параметра глубины от центра (заливки) к краям фигуры. При этом в виртуальном пространстве z-буфера изображение, например, прямоугольника будет выглядеть как усечённая пирамида, где верхняя грань соответствует заливке, а боковые наклонные грани — контуру. В случае «негативного» подслоя будет картина аналогичная, только инвертирована по глубине. Поскольку в «негативном» подслое фигуры определяют вырезы, для определения контуров необходимо расширить составляющую подслой слой геометрию на максимальную толщину контура в слое. При совмещении подслоёв в итоговое изображение, при наличии чего-либо в позитивном подслое и контура в негативном подслое, осуществляется проверка расстояния до границы фигуры, исходя из значения глубины, записанного в «негативном» подслое и параметрах толщины контура текущей фигуры из буфера с данными предыдущей стадии и максимальной толщины контура данного слоя. В итоговом изображении формируется точка контура или заливки в зависимости от результатов проверки (рис. 9).

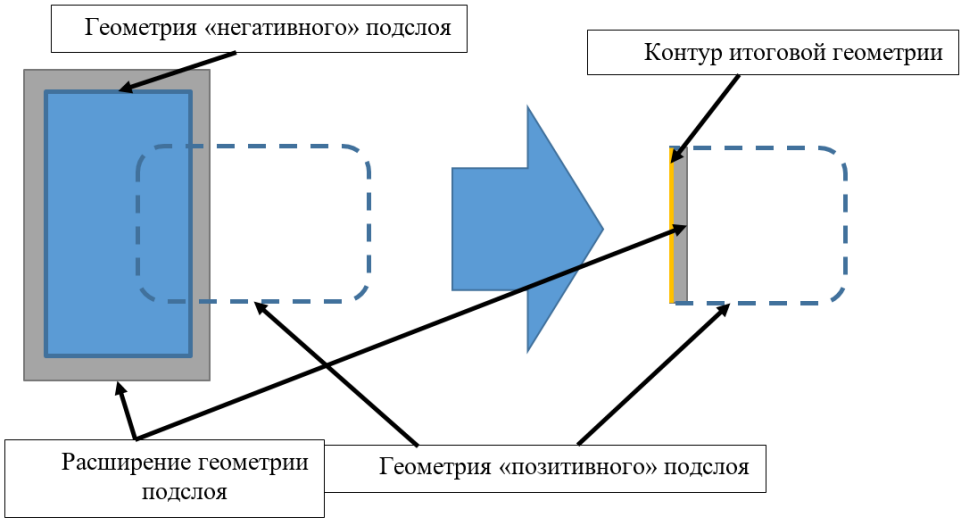

Рисунок 9 — Формирование контура фигуры с использованием Z-буфера

В листинге 1 представлен основной код шейдерной программы визуализации «позитивного» подслоя в форме псевдокода на основе языка GLSL. Описания форматов входных и выходных данных не являются существенно значимыми для работы алгоритма, и в листингах опущены. Отдельные этапы шейдерной программы (вершинный, геометрический и пиксельный) сведены в один листинг.

Листинг 1 — Шейдерная программа визуализации «позитивного» подслоя

```
void vertex_main()
{
  gl_Position = projectionMatrix*vec4(pos.x, pos.y, 0.5, 1.0);
  scr_Position = toScreenCoords(gl_Position.xy);
}

void fragment_main()
{
  bool outline = false;
  float realWidth = outlineWidth * scale * screenSpace;
  if (shape == SHAPE_POLYGON)
  {
    for (int i = 0; i < edges_num; ++i)
    {
      if (distance_to_edge(position, edges[i]) < realWidth)
        outline = true;
    }
  }
  else if (shape == SHAPE_CIRCLE)
  {
    float dist = distance(position, center);
    if (dist > radius || dist < innerRadius)
      discard;
    else if (dist > radius – realWidth || dist < innerRadius + realWidth)
      outline = true;
  }
  else
    discard;
  if (outline)
    color = outlineColor;
  else
    color = makeFill(fillType, scr_Position, fillColor);
}
```

Шейдерная программа №1 в зависимости от типа геометрии (многоугольник / окружность) осуществляет определение границы в экранных координатах и запись результатов вычисления цвета в результирующий буфер. В случае, если пиксель не является частью геометрии, он отбрасывается. Это происходит, например, при обработке окружностей, дуг и секторов, поскольку такие фигуры не могут быть представлены в виде примитивов графического процессора, и в реальности представляются многоугольниками (в данном решении — прямоугольниками) со специальным набором атрибутов.

Пропущенный в листинге 1 геометрический шейдер является тривиальным и просто копирует массивы параметров из входных в выходные. Тривиальные операции также опущены.

В листинге 2 представлена шейдерная программа визуализации «негативного» подслоя, также в форме псевдокода на основе GLSL. В ней используются две подпрограммы, не являющиеся стандартными:

- inflatePoly — выполняет расширение многоугольника на заданную величину, данная величина равна расстоянию от контура оригинального многоугольника до контура нового многоугольника. Поскольку в данной шейдерной программе обрабатываются исключительно треугольники (любые более сложные фигуры подвергаются триангуляции ранее), сложных циклов и нетривиальных особенностей подпрограмма не содержит.
- directed_distance_to_edge — вычисляет расстояние от точки до отрезка. Нетривиальным в данной функции является лишь случай, когда точка лежит за пределами фигуры — в таком случае возвращается расстояние со знаком «минус». Для реализации данного поведения, отрезки в массиве edges всегда направлены в одну сторону по порядку обхода многоугольника (т.е. всегда по либо против часовой стрелки).

Листинг 2 — Шейдерная программа визуализации «негативного» подслоя

```
void geometry_main()
{
  float realWidth = outlineWidth * scale * screenSpace;
  inflatePoly(edges, realWidth);
}
void fragment_main()
{
  float distance_to_edge = 0;
  if (shape == SHAPE_POLYGON)
  {
    for (int i = 0; i < edges_num; ++i)
    {
      float new_distance = directed_distance_to_edge(position, edges[i]);
          // возвращает отрицательное число, если точка снаружи многоугольника
      if (new_distance < 0)
        distance = min(distance_to_edge, new_distance);
    }
  }
  else if (shape == SHAPE_CIRCLE)
  {
    float dist = distance(position, center);
    if (dist > radius)
      distance_to_edge = radius - dist;
    else if (dist < innerRadius)
      distance_to_edge = dist – innerRadius;
  }
  else
    discard;
  if (distance_to_edge < 0)
    depth = 0.5 + distance_to_edge;
}
```

Шейдерная программа №2 для реализации последующего сведения с формированием контура (рис. 4) осуществляет специальную обработку контуров фигур. Если пиксель находится за контуром фигуры, расстояние до этого контура сохраняется в Z координате пикселя. Сохранение в буфере глубины данного значения требуется для корректной реализации операции объединения «негативной» геометрии. Буфер цвета в этой операции не участвует.

В листинге 3 приведена шейдерная программа сведения подслоёв, реализующая этап (стадию) №2 (см. список в начале раздела). Данная программа также представлена в форме псевдокода на основе GLSL, но содержит лишь один этап (пиксельный), поскольку остальные в данном случае являются тривиальными.

Листинг 3 — Шейдерная программа сведения подслоёв в слой

```
void fragment_main()
{
  color = vec4(0);
  for (int i  = 0; i < sublayers_num; ++i)
  {
    if (positive_sublayer(i))
      color += texture(sublayers[i].color, coordinates);
    else if (negative_sublayer(i))
    {
      float depth = texture(sublayers[i].depth, coordinates);
      if (depth >= 0.5)
        color = vec4(0);
      else
        color += outlineColor;
    }
  }
}
```

Шейдерная программа №3 выполняет сведение изображений подслоёв в изображение слоя. При этом происходит проверка параметра глубины негативных подслоёв, и, если она соответствует расширению

«негативной» геометрии, формируется граница. Если глубина соответствует основной части этой геометрии, предыдущее значение цвета данного пикселя аннулируется — формируется пустой фрагмент изображения.

Шейдерная программа №4 использует стандартные для аппаратно-ускоренной графики механизмы, поэтому его псевдокод не приводится.

Все шейдерные программы выполняются параллельно для отдельных фрагментов данных (вершин, примитивов, пикселей), в соответствии с архитектурными особенностями используемого графического процессора. Декомпозиция данных на фрагменты и объединение результатов расчётов отдельных фрагментов выполняется графическим процессором, что характерно для большинства реализованных на базе графического процессора алгоритмов визуализации [10].

Разработанная реализация была проверена на реальном проекте подложки современного процессора (фрагменты данного проекта приведены на рис. 2 и рис. 3), имеющим чуть более 1,6 млн. элементов. В таблице 1 приведены результаты замеров производительности алгоритма, выполненные на системе с центральным процессором Intel i5-11400H (базовая частота 2.7ГГц) и видеокартами Intel UHD Graphics, интегрированной в процессор, и дискретной nVidia GeForce RTX 3060 Laptop.

И интегрированные, и дискретные GPU выполняют шейдерные программы параллельно. Прямое исполнение шейдерных программ центральным процессором невозможно из-за различий в архитектурах графических и центральных процессоров. Программная имитация графического процессора силами центрального процессора низкопроизводительна, и сравнение с ней не имеет смысла.

Таблица 1 — Производительность типовых операций

	Intel UHD Graphics	nVidia GeForce RTX 3060
Загрузка файла	4278 мс, из них буферизация 4243 мс	4590 мс, из них буферизация 4515 мс
Прокрутка (смена области обзора)	1655 мс	20 мс
Масштабирование	1660 мс	18 мс
Добавление элемента	33 мс	30 мс
Удаление элемента	86 мс	77 мс

В случае использования интегрированных GPU и дискретных GPU нижнего ценового диапазона, скорость прорисовки находится ниже комфортного для пользователя уровня (в общем случае субъективное значение, в данной работе использовалась граница в 60 мс/кадр). В случае использования GPU среднего и верхнего ценового диапазона, скорость работы алгоритма обеспечивает комфортную работу.

Реализация прорисовки последовательным алгоритмом с использованием центрального процессора на том же проекте требует до нескольких десятков секунд на формирование изображения, в зависимости от количества видимых объектов.

Заключение

Предлагаемый набор из двух алгоритмов позволяет переложить большую часть задач по визуализации объектов в области САПР электронных устройств на высокопроизводительный графический процессор. При этом освобождается центральный процессор для выполнения других функции, и в то же время значительно увеличивается количество объектов, при котором сохраняется комфортное взаимодействие пользователя с системой. Данный алгоритм предполагает использование программно-аппаратного комплекса из центрального и выделенного графического процессора с соответствующим программным обеспечением. Данному требованию удовлетворяет уже большое количество распространённых персональных электронных вычислительных машин, что по факту обеспечивает высокую совместимость предложенного решения с уже существующими системами. В алгоритме реализованы определение контуров, работа с прозрачностью, контурами и параметрическими заливками. Реализация алгоритма успешно справляется с задачами, характерными для текущего уровня развития отрасли (1-2 млн элементов) обеспечивая субъективно комфортную скорость работы. Предполагается использовать разработанный алгоритм в новом поколении САПР электронных устройств с характерным размером задачи в единицы миллионов элементов.

Список литературы

1. Шамаев Е.А., Барановский А.О., Бережной А.А., Матюшин П.М. Проблемы и перспективы развития технологии чиплетов // Наноиндустрия. 2020. Т.13. №99. С.112-114

2. Бондарев В.Н., Воскресенский А.А. Анализ эффективности сапр для моделирования тепловых процессов в печатных платах // Сапр и моделирование в современной электронике: Сборник научных трудов IV Международной научно-практической конференции. Брянск. 2020. С.17-19

3. Мальцев А.В. Методы распределенной визуализации тумана в трехмерной виртуальной среде с использованием GPU // Информационные технологии и вычислительные системы. 2020. №3. С.62-70

4. Тимохин П.Ю., Михайлюк М.В. Компьютерное моделирование и визуализация точных ландшафтных теней в системе виртуальной среды // Научная визуализация. 2022. Т.14. №2. С.77-87

5. Пэйн Д. [Payne D.] Использование GPU для ускорения редактирования топологии печатных плат [Электронный ресурс] // SemiWiki Открытый форум для профессионалов в области полупроводников. 2022. URL: https://semiwiki.com/eda/cadence/308551-using-a-gpu-to-speed-up-pcb-layout-editing/ (Дата обращения 01.03.2023)

6. Лу Ю. [Yifan Lu], Янг Л. [Lu Yang], Бхавсар В. [Virendrakumar C. Bhavsar], Кумар Н. [Neetesh Kumar] Обработка древовидных структур данных на графических процессорах // 7-я международная конференция по облачным вычислениям, науке о данных и инженерии. Ноида, 2017. doi: 10.1109/CONFLUENCE.2017.7943203

7. Лефон А. [Lefohn A.], Книсс Д. [Kniss J.], Овенс Д. [Owens J.] Реализация эффективных параллельных структур данных на графических процессорах // GPU gems 2: Техники программирования для высокопроизводительных графических и вычислений общего назначения; под ред. Фарр М. [Pharr M.] ISBN 0-321-33559-7 Гл.33

8. Тарасов И.Е., Мирзоян Д.И., Лобанов И.Н., Федоткин А.С. Архитектура САПР для проектирования корпусов СБИС // Высокопроизводительные вычислительные системы и технологии. 2022. Т.6. №1. С.13-18.

9. Работа с представлением сборочного чертежа печатной платы в Altium Designer [Электронный ресурс] // Документация Altium Designer. 2023. URL: https://www.altium.com/ru/documentation/altium-designer/working-with-draftsman-board-assembly-view (Дата обращения 02.04.2023)

10. Лебешков Д.А. Обзор работы шейдерных программ и графического конвейера в DirectX // Наука, образование, инновации: актуальные вопросы и современные аспекты сборник статей X Международной научно-практической конференции. Пенза. 2021. С.23-25

References

1. Shamaev E.A., Baranovskij A.O., Berezhnoj A.A., Matjushin P.M. Problemy i perspek-tivy razvitija tehnologii chipletov (Problems and prospects for the development of chiplet technology) // Nanoindustrija. 2020. vol.13. no.99. pp.112-114 (in Russian)

2. Bondarev V.N., Voskresenskij A.A. Analiz jeffektivnosti sapr dlja modelirovanija teplovyh processov v pechatnyh platah (CAD efficiency analysis for simulation thermal processes in PCBs) // Sapr i modelirovanie v sovremennoj jelektronike: Sbornik nauchnyh trudov IV Mezhdunarodnoj nauchno-prakticheskoj konferencii. Brjansk. 2020. pp.17-19 (in Russian)

3. Mal'cev A.V. Metody raspredelennoj vizualizacii tumana v trehmernoj virtual'noj srede s ispol'zovaniem GPU (Methods for distributed visualization of fog in 3d virtual environment using GPU)// Informacionnye tehnologii i vychislitel'nye sistemy. 2020. no.3. pp.62-70 (in Russian)

4. Timokhin P.Yu., Mikhaylyuk M.V. Computer modelingand visualization of accurate terrain shadows in virtual environment system // Nauchnaja vizualizacija. 2022. vol.14. no.2. pp.77-87

5. Payne D. Using a GPU to Speed Up PCB Layout Editing [Электронный ресурс] // SemiWiki The open forum for semiconductor professionals. 2022. URL: https://semiwiki.com/eda/cadence/308551-using-a-gpu-to-speed-up-pcb-layout-editing/ (Дата обращения 01.03.2023)

6. Yifan Lu, Lu Yang, Virendrakumar C. Bhavsar, Neetesh Kumar Tree structured data processing on GPUs // 2017 7th International Conference on Cloud Computing, Data Science & Engineering - Confluence. Noida, India, 2017. doi: 10.1109/CONFLUENCE.2017.7943203

7. Lefohn A., Kniss J., Owens J. Implementing Efficient Parallel Data Structures on GPUs // GPU gems 2: programming techniques for high-performance graphics and general-purpose computation; / edited by Matt Pharr ISBN 0-321-33559-7 Ch.33

8. Tarasov I.E., Mirzoyan D.I., Lobanov I.N., Fedotkin A.S. Arhitektura SAPR dlja proektirovanija korpusov SBIS (CAD architecture for design of VLSI packages) // Vysokoproizvoditel'nye vychislitel'nye sistemy i tehnologii. 2022. vol.6. no.1. pp.13-18

9. Working with the Draftsman Board Assembly View in Altium Designer [Электронный ресурс] // Altium Designer Documentation. 2023. URL: https://www.altium.com/ru/documentation/altium-designer/working-with-draftsman-board-assembly-view (Дата обращения 02.04.2023)

10. Lebiashkou D.A. Obzor raboty shejdernyh programm i graficheskogo konvejera v DirectX (Overview of the work of shader programs and the graphics pipeline in DirectX) // NAUKA, OBRAZOVANIE, INNOVACII: AKTUAL'NYE

VOPROSY I SOVREMENNYE ASPEKTY sbornik statej X Mezhdunarodnoj nauchno-prakticheskoj konferencii. Penza. 2021. pp.23-25