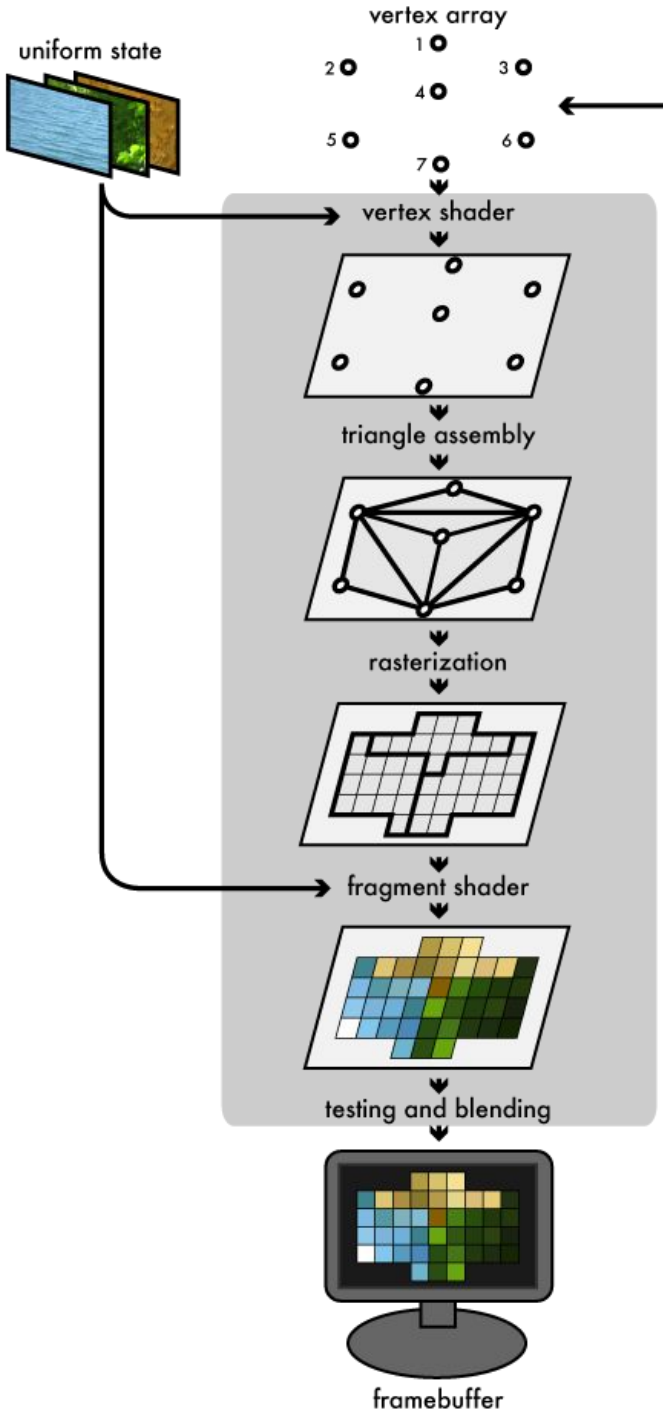
Растеризация: OpenGL, Larrabee, cudaraster

Вычисления на видеокартах. Лекция 10

1. **OpenGL**: hardware rasterization
2. Software rasterization on **CPU and Intel Larrabee**
3. Software rasterization on **GPGPU**

Полярный Николай

polarnick239@gmail.com



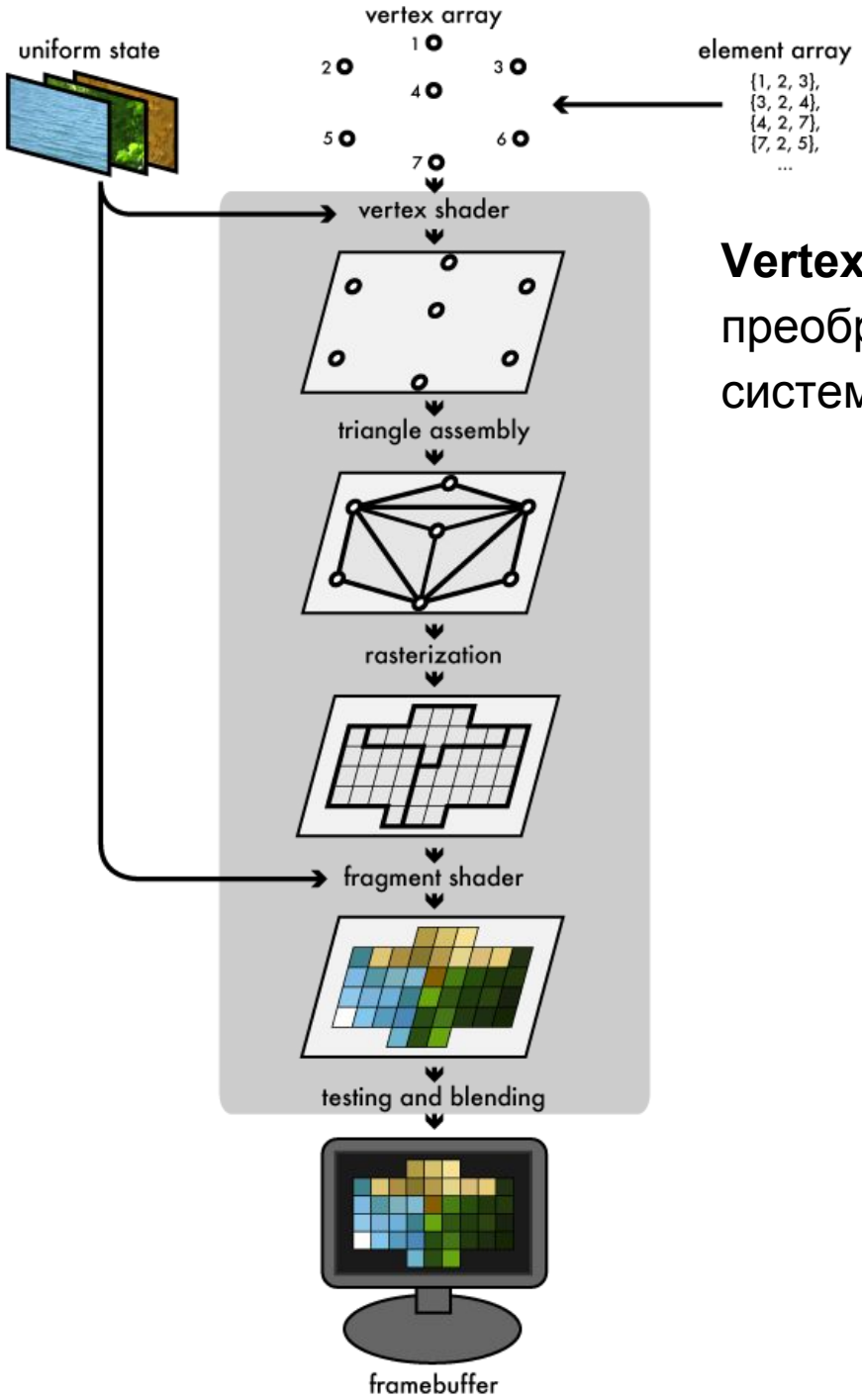
OpenGL

Vertex array - массив упорядоченных координат точек пространства. Но vertex attributes могут также включать произвольную информацию о вершинах, например текстурные *uv*-координаты.

Element array - описания геометрических примитивов. Треугольник описывается тройкой индексов вершин на которые он опирается.

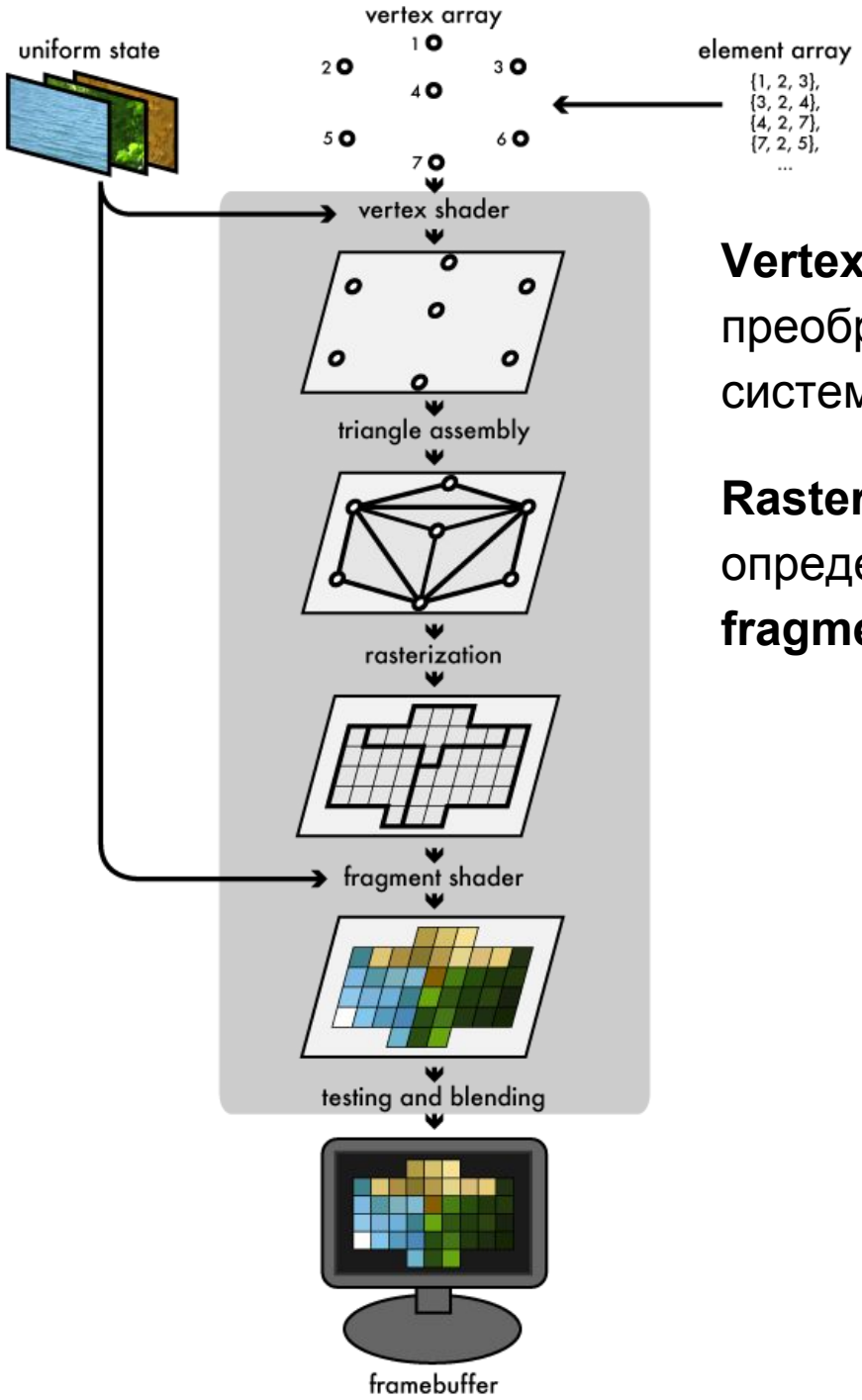
Uniform state - текстуры, информация об освещении сцены и т.п..

Подробнее:
http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html



OpenGL

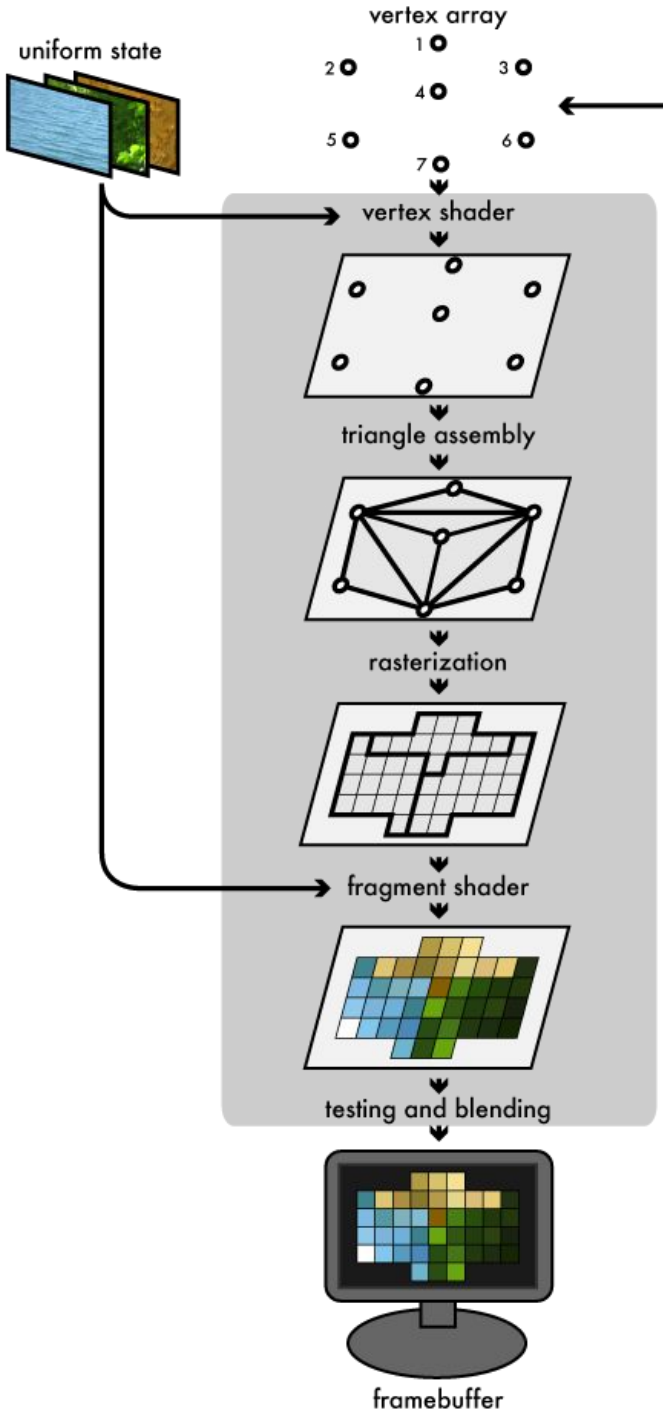
Vertex Shader: любые по-вершинные преобразования (в первую очередь - переход в систему координат экрана).



OpenGL

Vertex Shader: любые по-вершинные преобразования (в первую очередь - переход в систему координат экрана).

Rasterization: для каждого треугольника определить множество покрытых пикселей (т.н. **fragments**).

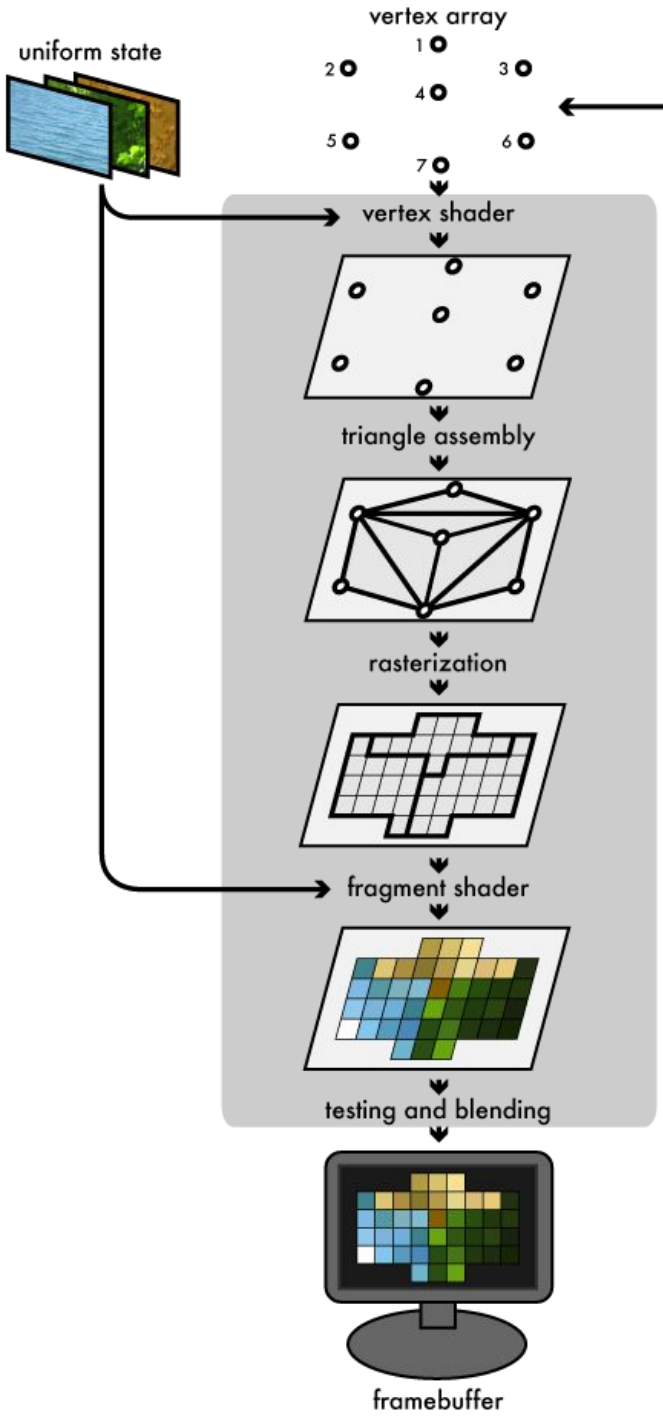


OpenGL

Vertex Shader: любые по-вершинные преобразования (в первую очередь - переход в систему координат экрана).

Rasterization: для каждого треугольника определить множество покрытых пикселей (т.н. **fragments**).

Fragment Shader: любые по-пиксельные (per-fragment) преобразования. Например накладывание цвета из текстуры, расчет эффектов освещения, расчет глубины фрагмента и т.п..



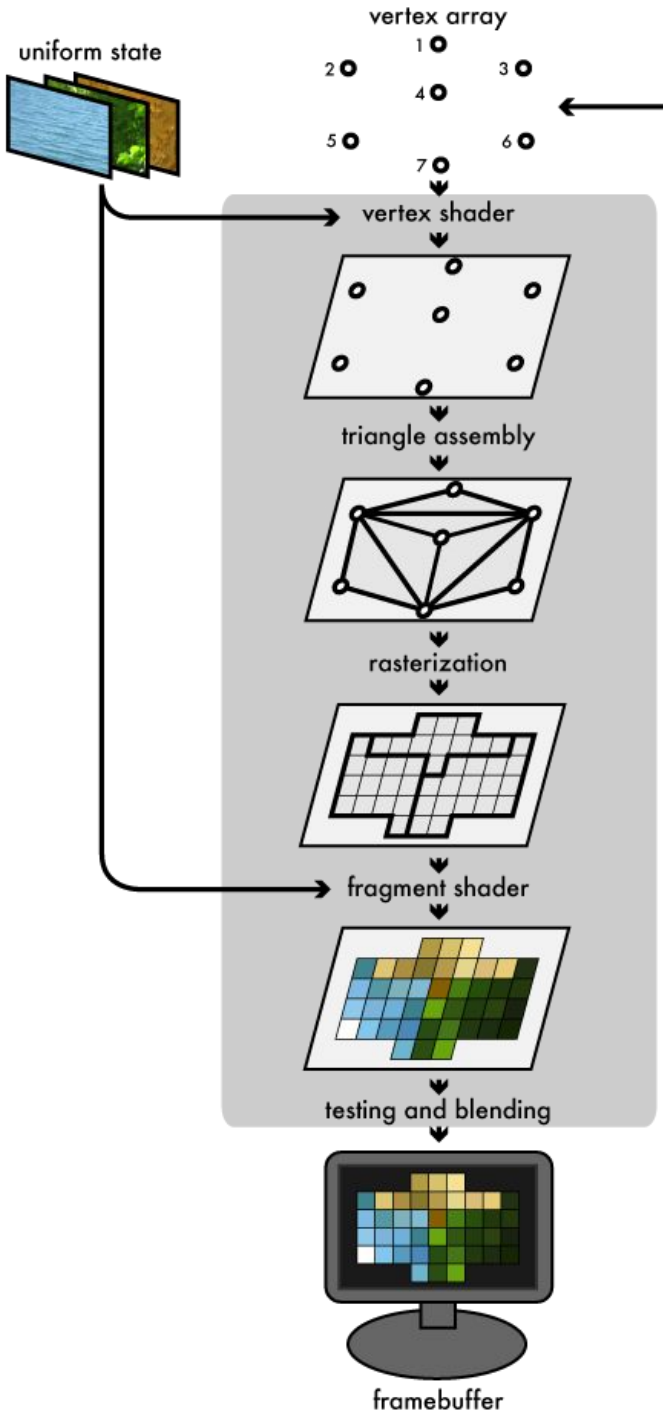
OpenGL

Vertex Shader: любые по-вершинные преобразования (в первую очередь - переход в систему координат экрана).

Rasterization: для каждого треугольника определить множество покрытых пикселей (т.н. **fragments**).

Fragment Shader: любые по-пиксельные (per-fragment) преобразования. Например накладывание цвета из текстуры, расчет эффектов освещения, расчет глубины фрагмента и т.п..

Testing and blending: в каждом пикселе остается цвет фрагмента с наименьшей глубиной. Альтернативно возможно ограниченное смешение цветов фрагментов.



OpenGL

Vertex Shader (programmable): любые по-вершинные преобразования (в первую очередь - переход в систему координат экрана).

Rasterization (hardware): для каждого треугольника определить множество покрытых пикселей (т.н. **fragments**).

Fragment Shader (programmable): любые по-пиксельные (per-fragment) преобразования. Например накладывание цвета из текстуры, расчет эффектов освещения, расчет глубины фрагмента и т.п..

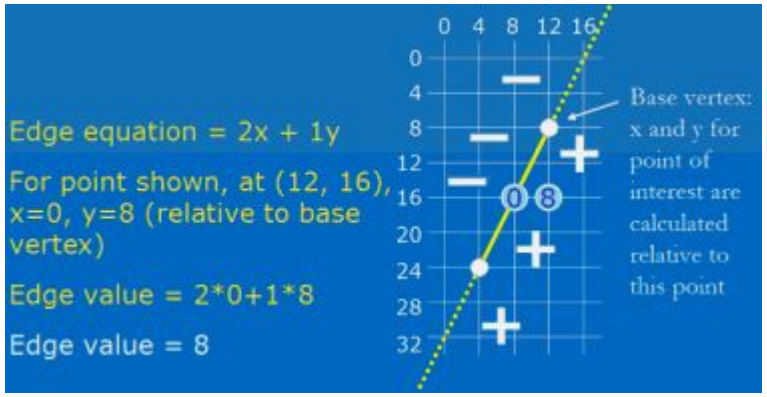
Testing and blending (hardware): в каждом пикселе остается цвет фрагмента с наименьшей глубиной. Альтернативно возможно ограниченное смешение цветов фрагментов.

1) Алгоритм Брезенхэма - растеризация отрезка

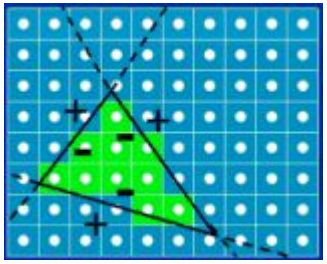
```
void line(x0, y0, x1, y1) {
    bool steep = false;
    if (abs(x0 - x1) < abs(y0 - y1)) { // if the line is steep, we transpose the image
        swap(x0, y0);
        swap(x1, y1);
        steep = true;
    }
    if (x0 > x1) { // make it left-to-right
        swap(x0, x1);
        swap(y0, y1);
    }

    for (int x = x0; x <= x1; ++x) {
        float t = (x - x0) / (x1 - x0);
        int y = y0 * (1 - t) + y1 * t;
        if (steep) {
            put_pixel(y, x); // if transposed, de-transpose
        } else {
            put_pixel(x, y);
        }
    }
}
```

Подробнее: [habrahabr - пишем упрощённый OpenGL своими руками]

2) CPU: brute force перебор



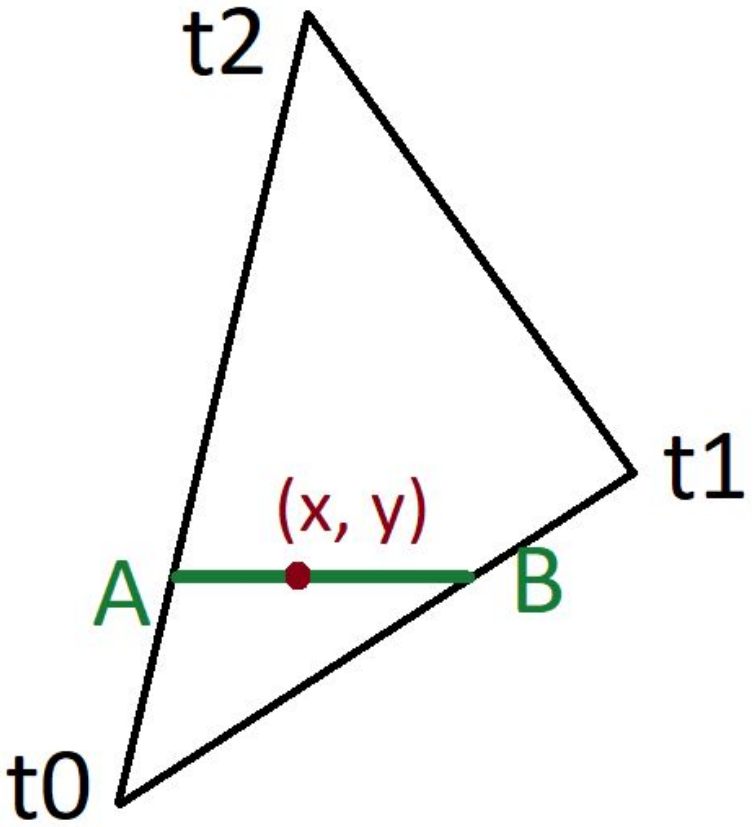

```
void rasterize(Point t0, Point t1, Point t2) {
    bbox = find_bounding_box(t0, t1, t2);
    for (each pixel in bbox) {
        if (inside(t0, t1, t2, pixel)) {
            put_pixel(pixel);
        }
    }
}
```



Подробнее: [habrahabr - пишем упрощённый OpenGL своими руками]

2) CPU: аккуратный перебор

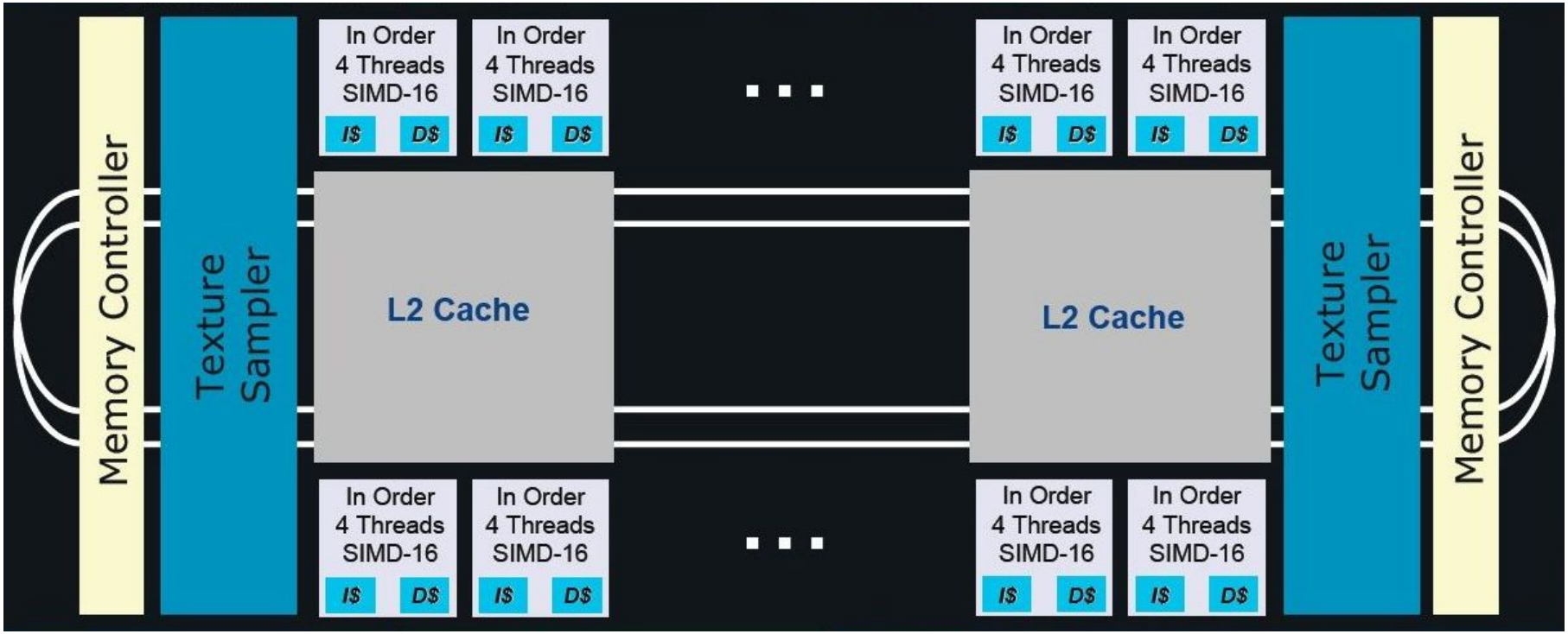
```
void rasterize(Point t0, Point t1, Point t2) {
    // sort the vertices, t0, t1, t2 lower-to-upper
    if (t0.y > t1.y) swap(t0, t1);
    if (t0.y > t2.y) swap(t0, t2);
    if (t1.y > t2.y) swap(t1, t2);

    int total_height = t2.y - t0.y;
    for (int y = t0.y; y < t1.y; ++y) {
        float segment_height = t1.y - t0.y;
        float alpha = (y - t0.y) / total_height;
        float beta  = (y - t0.y) / segment_height;
        Point A = t0 + (t2 - t0) * alpha;
        Point B = t0 + (t1 - t0) * beta;
        if (A.x > B.x) swap (A, B);
        for (int x = A.x; x <= B.x; ++x) {
            put_pixel(x, y);
        }
    }
    for (int y = t1.y; y <= t2.y; ++y) {
        ...
    }
}
```



Подробнее: [habrahabr - пишем упрощённый OpenGL своими руками]

Larrabee, 2008

# CPU cores:	2 out-of-order	10 in-order
Instruction issue:	4 per clock	2 per clock
VPU per core:	4-wide SSE	16-wide
L2 cache size:	4 MB	4 MB
Single-stream:	**4 per clock**	**2 per clock**
Vector throughput:	**8 per clock**	**160 per clock**

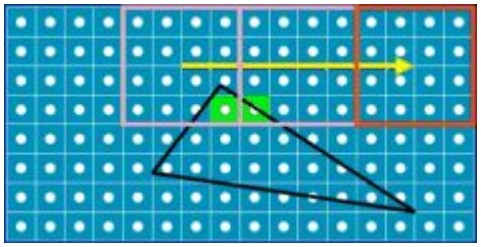


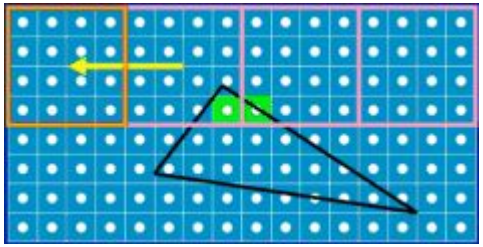
3) Larrabee: Sweep Rasterization

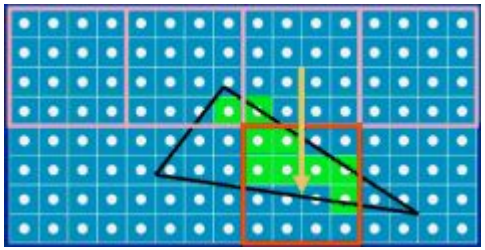
Ради хорошей векторизуемости - обработка **блоком 4x4** (16 - ширина SIMD).

Начав с верхней вершины треугольника:

1) Скользим вправо, пока треугольник не закончится.
2) Скользим влево, пока треугольник не закончится.
3) Делаем шаг вниз и повторяем процедуру.



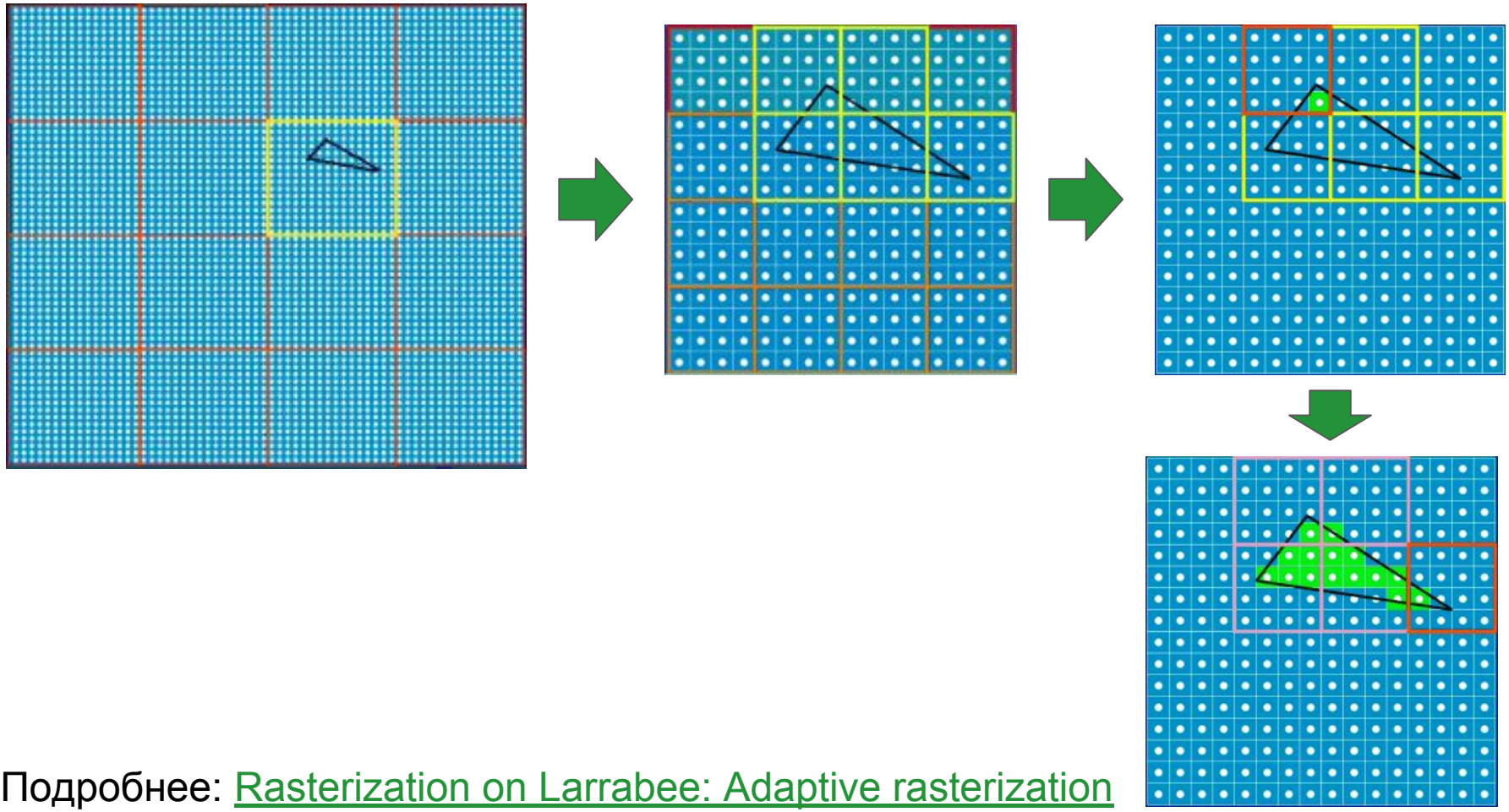




Подробнее: [Rasterization on Larrabee: Why Rasterization Was the Problem Child]

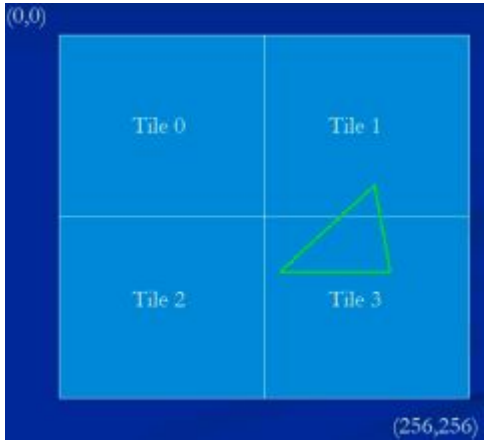
4) Larrabee: Hierarchical approach (для 64x64)

64x64 Tile: двухуровневый спуск по **4x4 решетке** (16 - ширина SIMD):



Подробнее: [Rasterization on Larrabee: Adaptive rasterization]

4) Larrabee: Tile Assignment



Как проверить какие **tiles** треугольник пересекает?

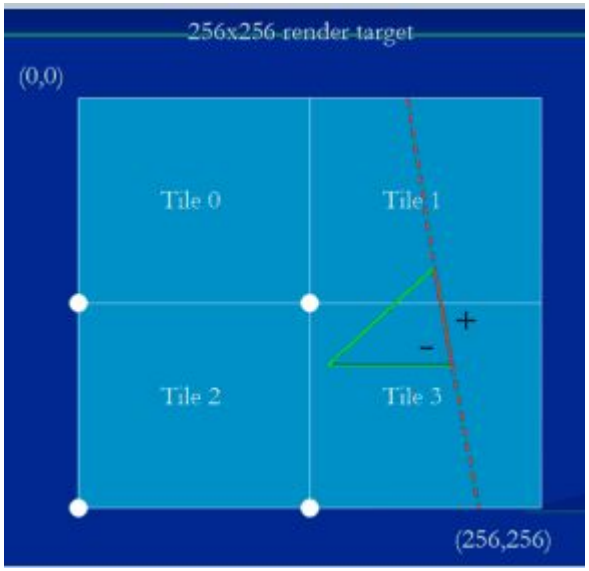
Подробнее: Rasterization on Larrabee: Adaptive rasterization

4) Larrabee: Tile Assignment - **trivial reject test**

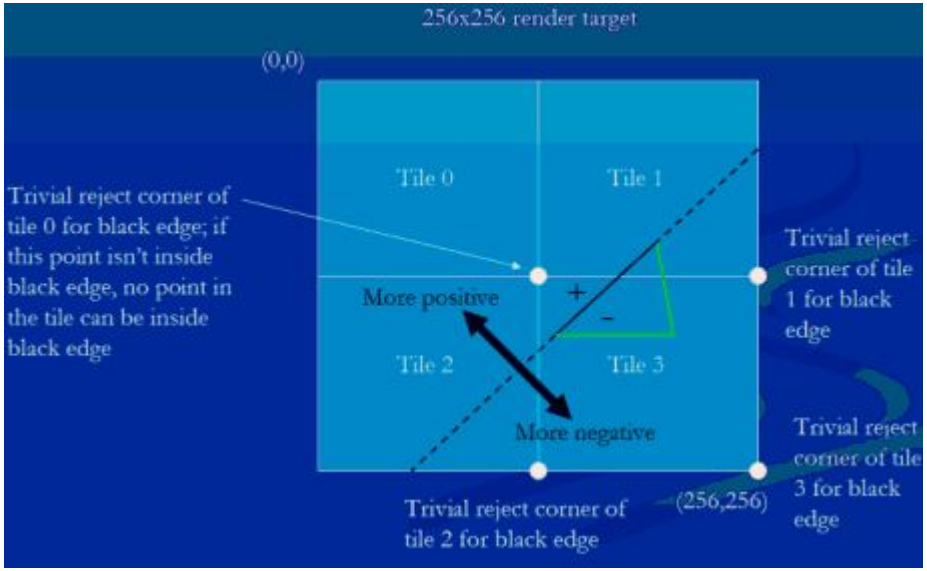
Trivial reject corner - угол **Tile** с наиболее **отрицательным** результатом уравнения прямой (наиболее внутри).

В разных случаях разные **trivial reject corner**.

Все - нижние левые:



Все - нижние правые:



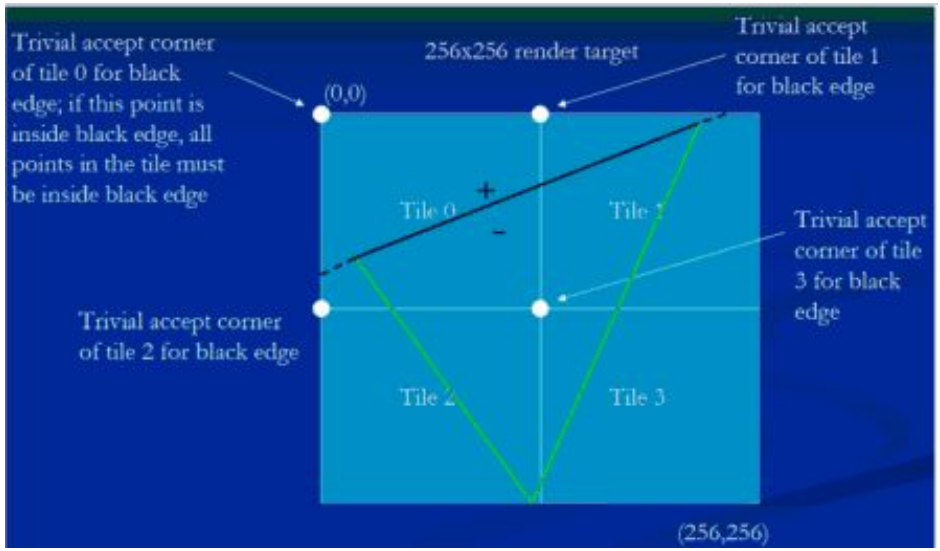
Подробнее: [Rasterization on Larrabee: Adaptive rasterization]

4) Larrabee: Tile Assignment - **trivial accept test**

Trivial accept corner - угол **Tile** с наиболее **положительным** результатом уравнения прямой (наиболее снаружи).

Trivial accept corner лежит напротив **trivial reject corner**.

Tile 2 и Tile 3 проходят trivial accept test:

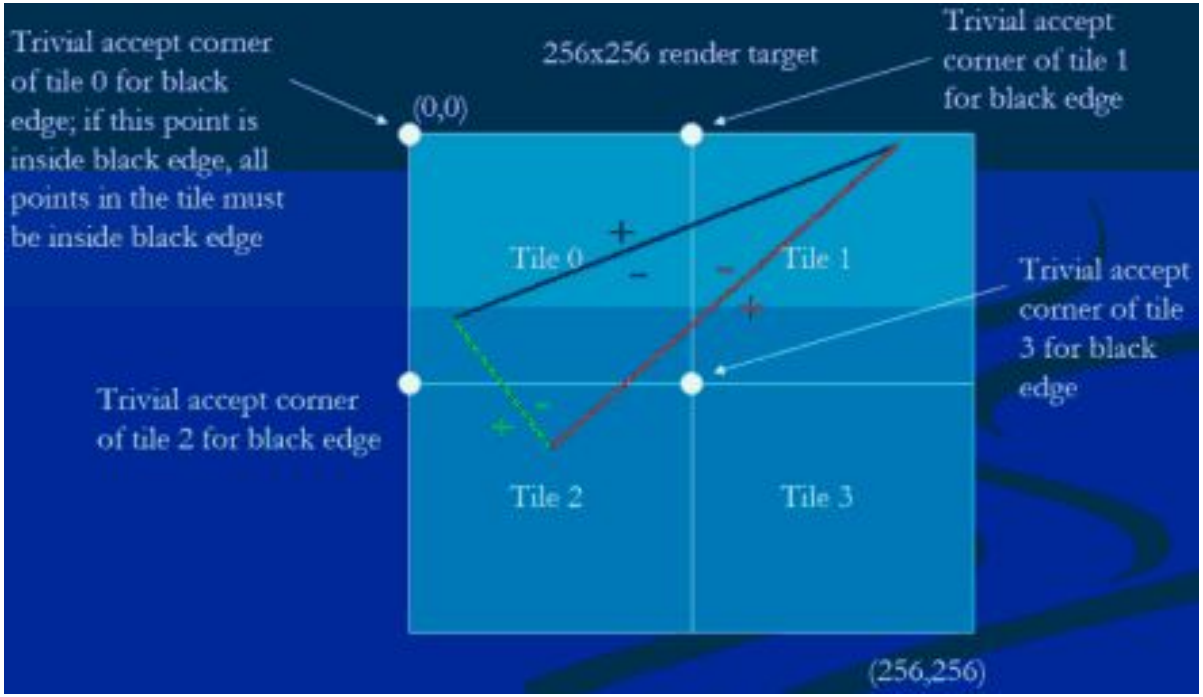


Подробнее: [Rasterization on Larrabee: Adaptive rasterization]

4) Larrabee: Tile Assignment

Если tile проходит **trivial reject test** относительно одной из сторон треугольника - треугольник не пересекает tile.

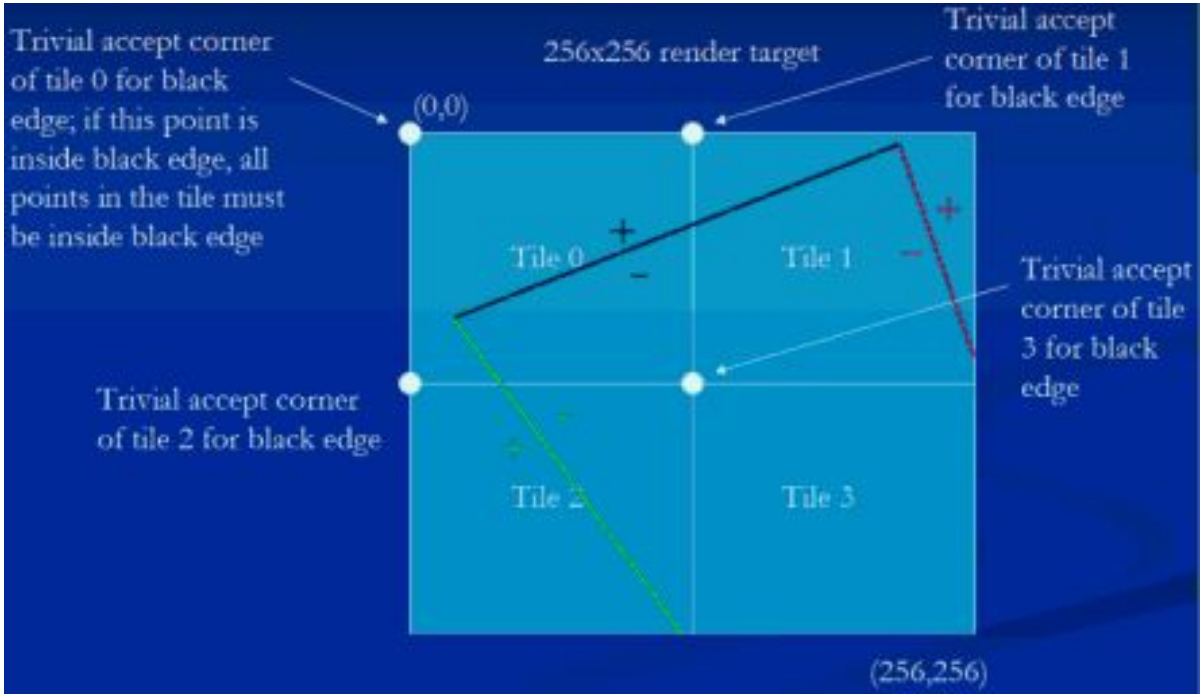
Если tile проходит **trivial accept test** относительно одной из сторон треугольника - только относительно этой стороны не нужны дальнейшие проверки, но про остальные стороны ничего неизвестно:



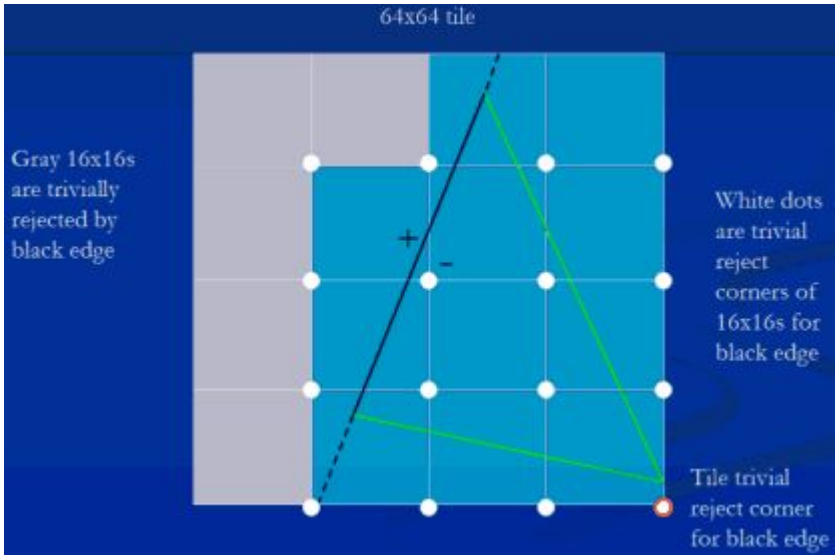
4) Larrabee: Tile Assignment

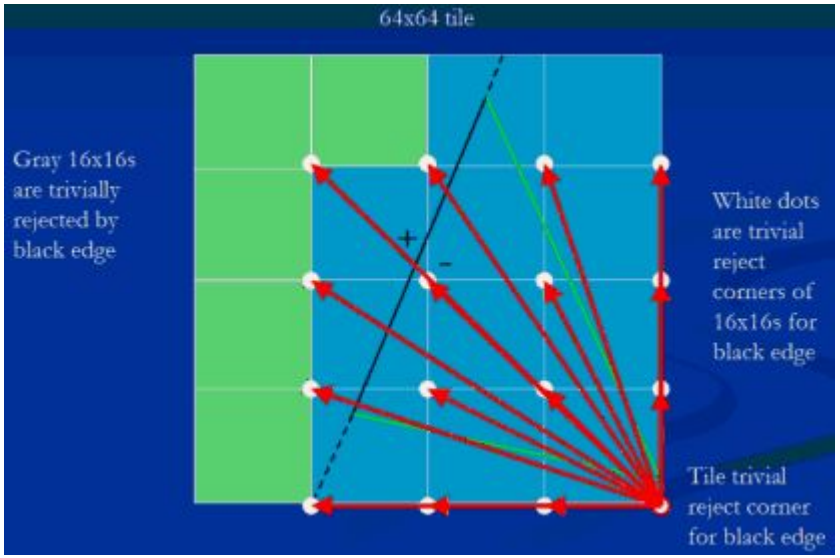
Но иногда дарят подарок!

Иногда tile проходит **trivial accept test** относительно **всех** трех ребер:



4) Larrabee: trivial accept/reject test (для 64x64)





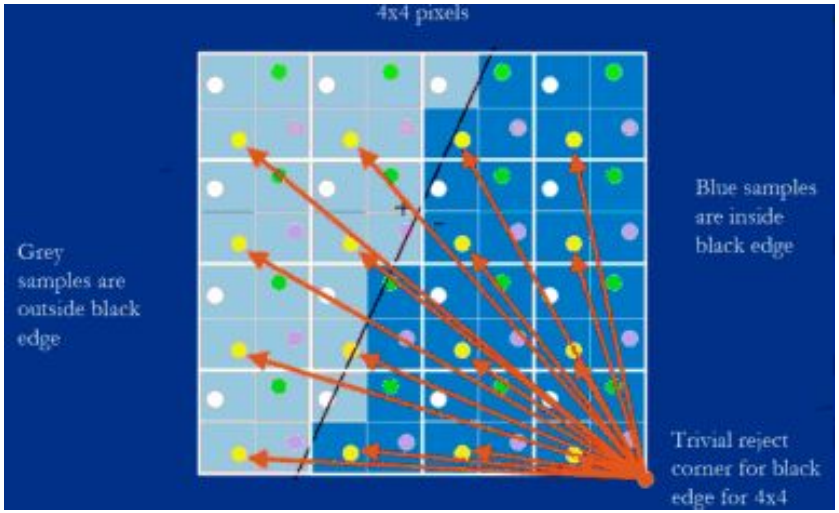
Аналогично проводим оба вида тестов чтобы выкинуть лишнюю работу или чтобы понять что тайл целиком покрыт треугольником и т.о. растеризацией покрыт целиком. (на по-пиксельном уровне уже нужен лишь trivial reject test)

Серые тайлы - отсеклись черной стороной по **trivial reject test**.

Идеально ложится на SIMD (тривиальная арифметика и битовые операции).

Подробнее: Intra-tile Rasterization: 16×16 Blocks

4) Larrabee: MSAA (multisample antialiasing)



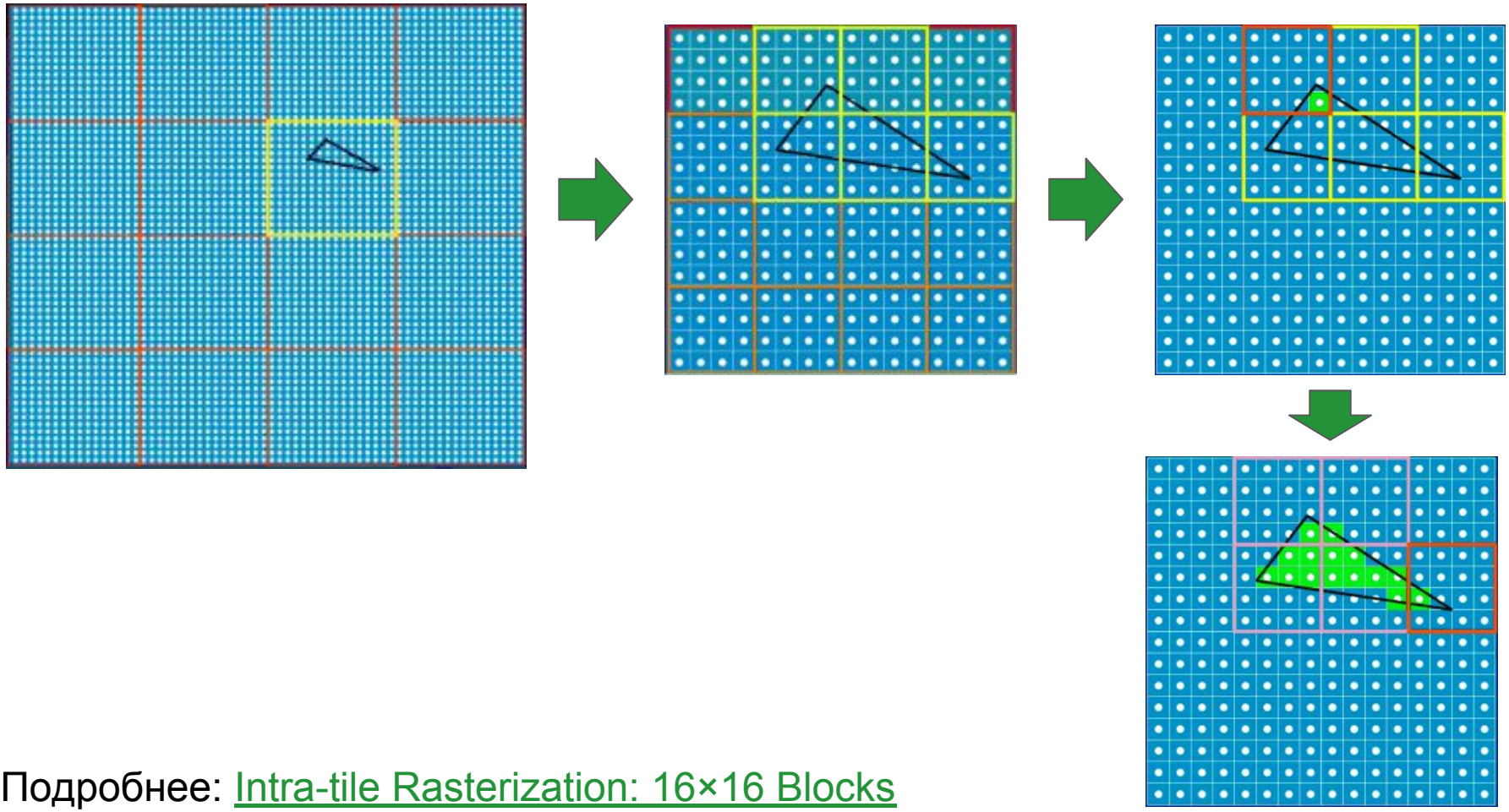
Для trivially accepted tiles - нет дополнительных операций.

Для partially accepted 4x4 tiles - просто еще один прогон trivial reject corner по 16 сэмплов.

Подробнее: Intra-tile Rasterization: 16×16 Blocks

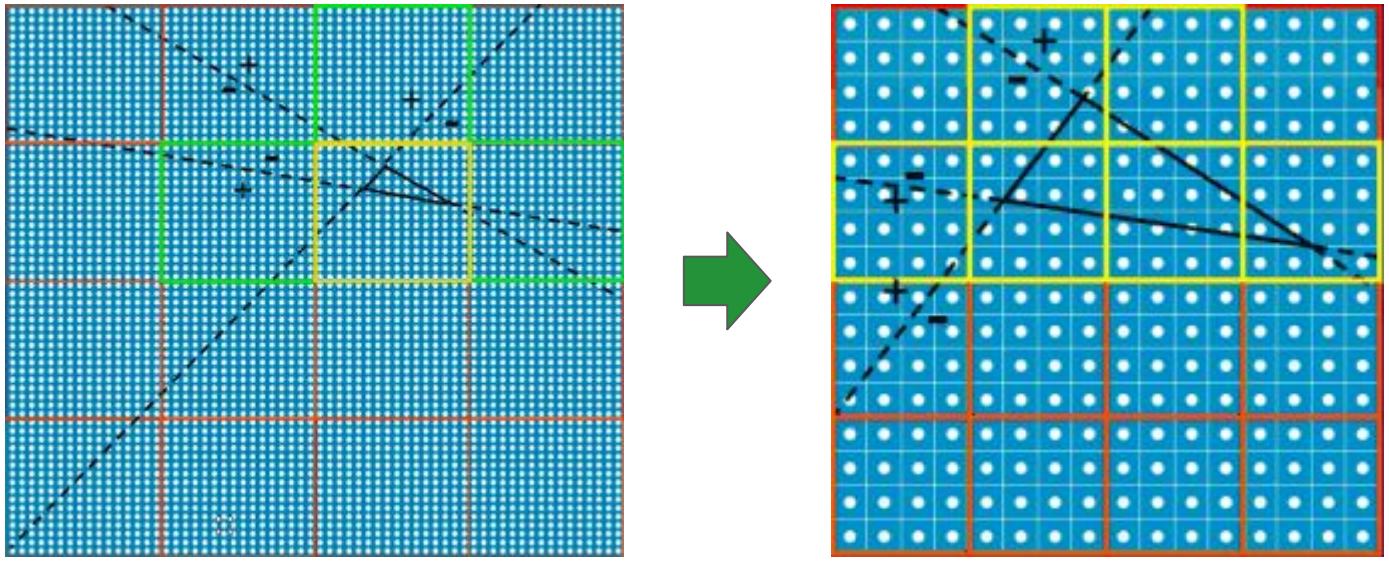
4) Larrabee: Hierarchical approach (для 64x64)

64x64 Tile: двухуровневый спуск по **4x4 решетке** (16 - ширина SIMD):



Подробнее: Intra-tile Rasterization: 16×16 Blocks

4) Larrabee: Level 1



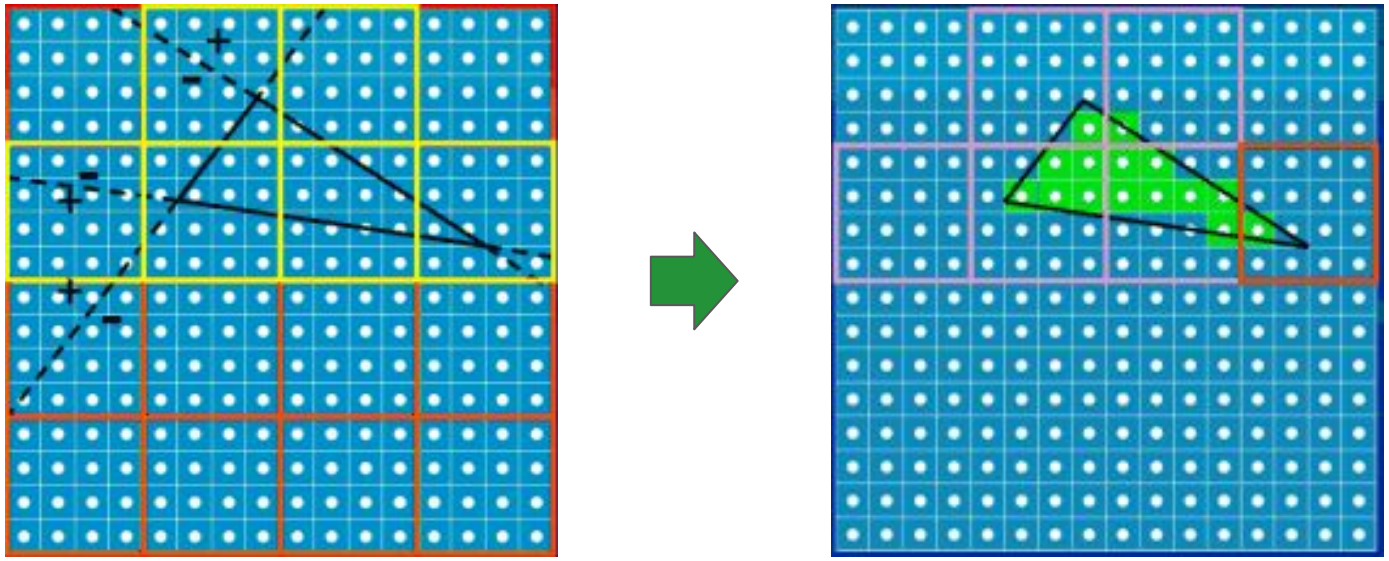
Есть 4x4 сетка из 64x64 тайлов.

12 тайлов отсеклись по trivial reject test.
(trivial accept test случается только для больших треугольников)

Спускаемся в оставшиеся 4 тайла.

Подробнее: Larrabee: Putting It All Together

4) Larrabee: Level 2



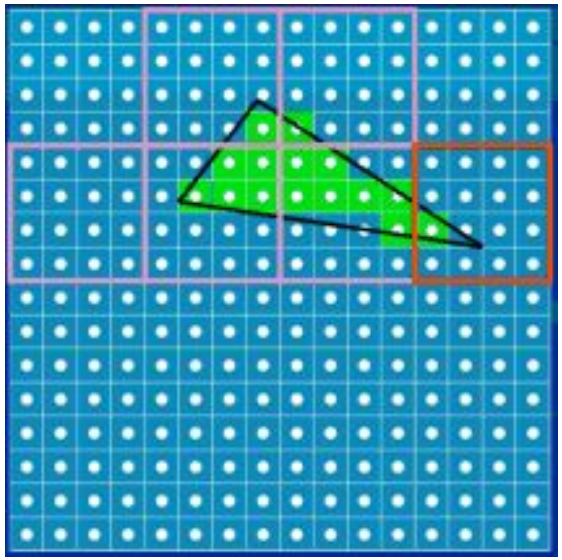
Есть 4x4 сетка из 16x16 тайлов.

10 тайлов отсеклись по trivial reject test.

Спускаемся в оставшиеся 6 тайлов.

Подробнее: Larrabee: Putting It All Together

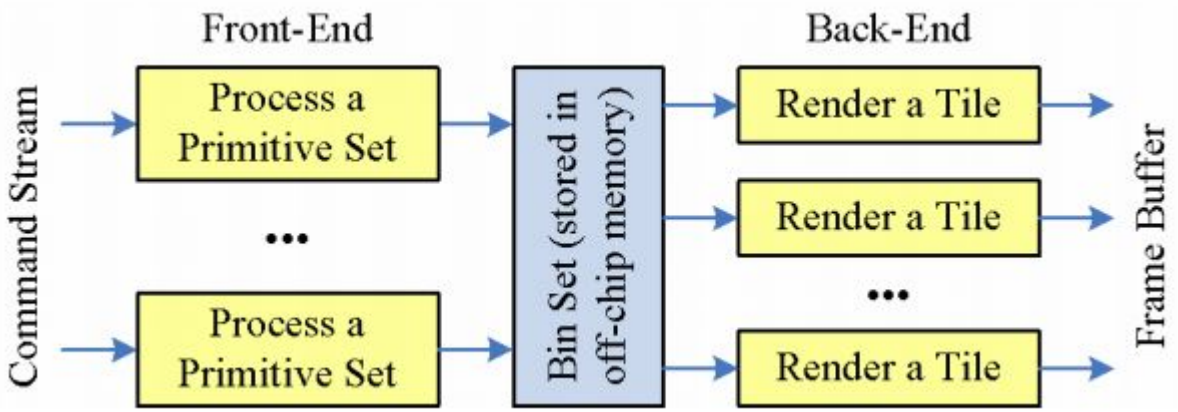
4) Larrabee: Level 3



Наконец, в блоках из 4x4 пикселей считаем маски.

Подробнее: Larrabee: Putting It All Together

4) Larrabee



1) Для каждого треугольника находятся пересекающиеся корзины-тайлы и для них вычисляются маски покрытия (или флажок **trivial accept**).
2) В каждой корзине для всех треугольников выполняется фрагментный шейдер.

Размер корзины таков, чтобы ее tile влезал в L2 cache (128x128 для 256 KB).

Подробнее: Larrabee: A Many-Core x86 Architecture for Visual Computing

4) Larrabee

Ссылки:

- Dr.Dobb’s: Rasterization on Larrabee
- Larrabee: A Many-Core x86 Architecture for Visual Computing
- Why didn't Larrabee fail?

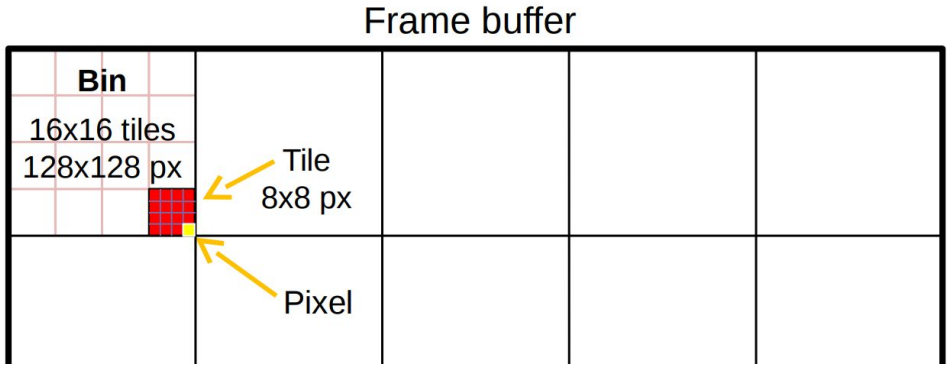
5) cudaraster: software rasterization on GPGPU

Triangle Setup → **Bin Raster** → **Coarse Raster** → **Fine Raster**

Нужно:

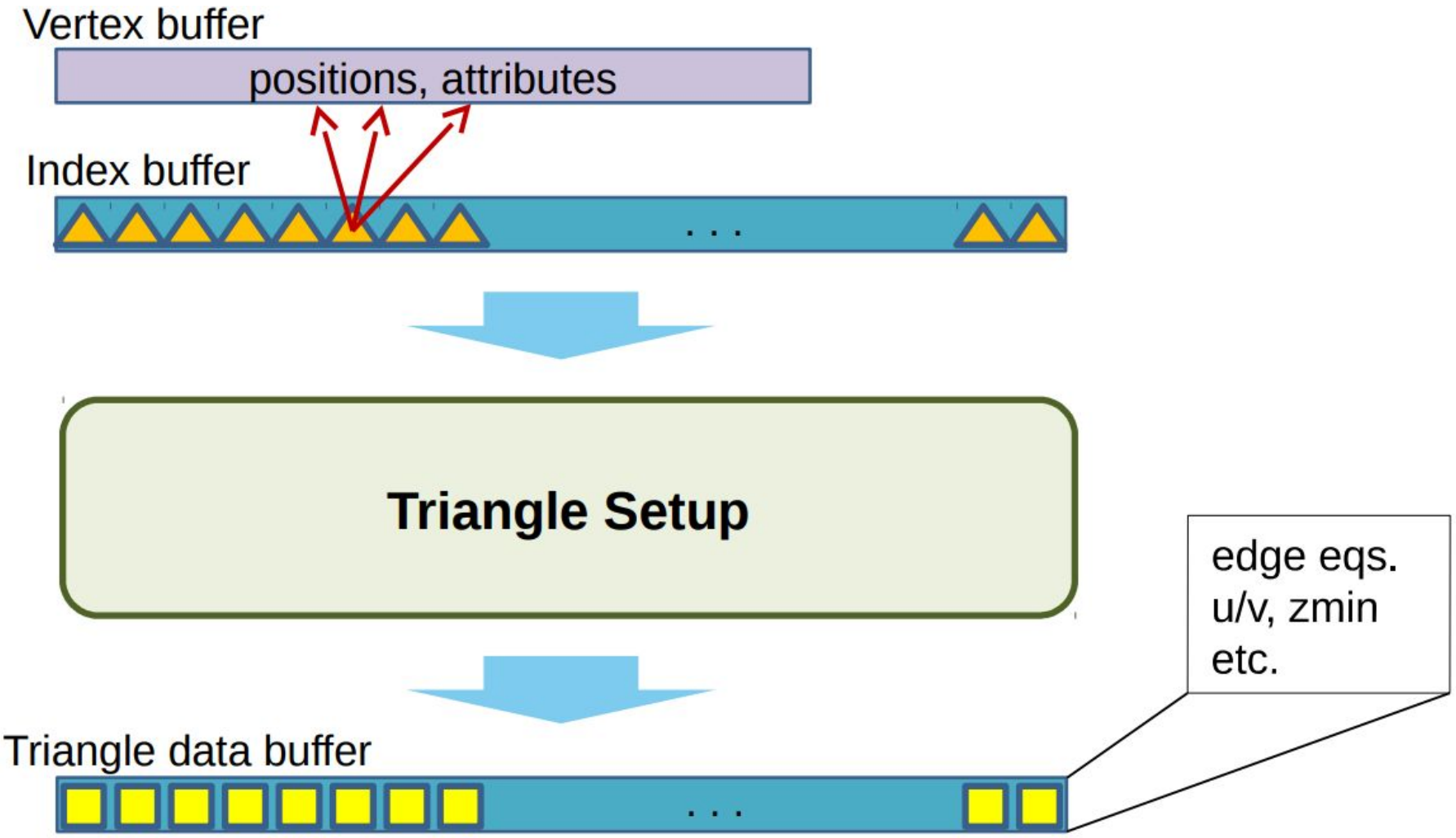
- Массовый параллелизм. Хорошая гранулярность и балансировка рабочей нагрузки.
- Минимизировать число синхронизаций между Streaming Multiprocessor (**SM**, в статье упоминаются как **CTA** = **C**ooperative **T**hread **A**rrays). Т.е. минимизировать число глобальных атомарных операций.

Общая идея все та же - распределить треугольники по корзинам и обработать по тайлам:



Подробнее: cudaraster

5) cudaraster: Triangle Setup

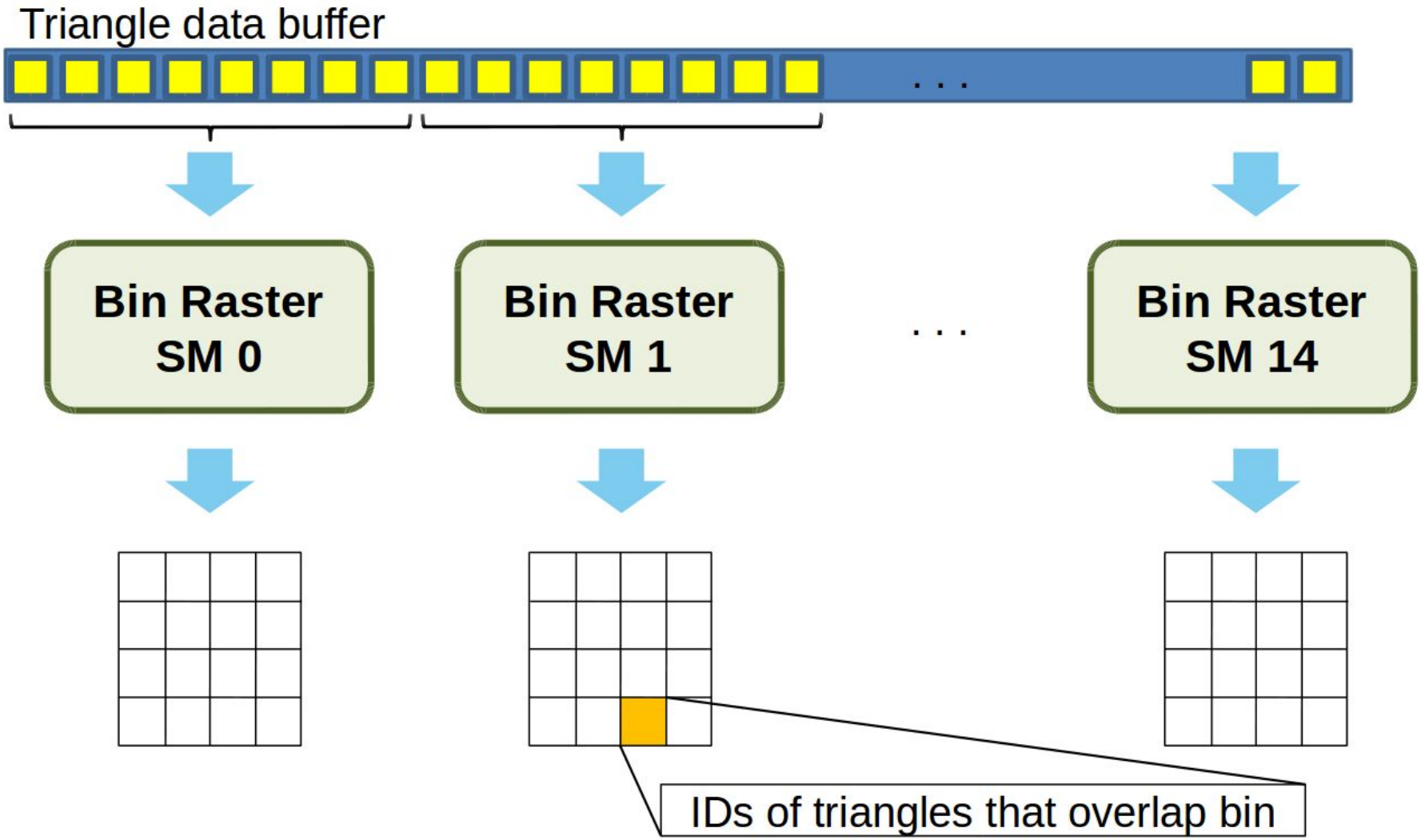


5) cudaraster: Triangle Setup

Множество culling tests для каждого треугольника:

- Если площадь ноль - выкинули
- Если площадь отрицательная (если backface culling) - выкинули
- Если bounding box между рядов семплов - выкинули
  (тонкие вертикальные/горизонтальные - например наблюдаемые под прямым углом стены вдалеке)
- Если AABB достаточно мал чтобы в нем было всего пара семплов

5) cudaraster: Bin Raster



5) cudaraster: Bin Raster **SM** - First Phase

1) **SM** бронирует очередной пакет треугольников (**atomic**, 16K треугольников)
2) Забирает из пакета часть - 512 треугольников
3) Выполняет **culling**/**clipping** и в локальной памяти префиксными суммами определяет сколько треугольников эффективно получилось
4) Пока треугольников эффективно оказывается мало - повторять шаги

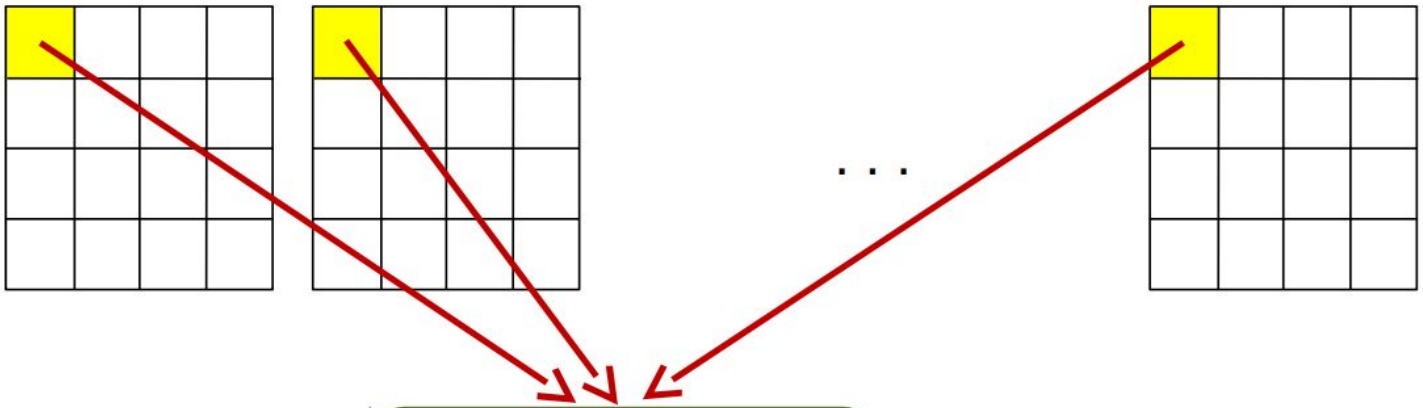
5) cudaraster: Bin Raster **SM** - Second Phase

Когда треугольников подгружено достаточно для утилизации всех потоков - каждый треугольник обрабатывается своим потоком:

- Определяются **корзины** пересекающие треугольник
- Оптимизированный код для случая треугольника 2x2 пикселя и меньше

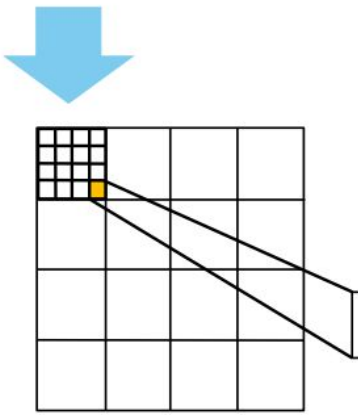
Результат пишется в специально выделенную для данного SM очередь (соответственно не нужна синхронизация между разными SM)

5) cudaraster: Coarse Raster



Coarse Raster
SM *n*

На каждую корзину
назначается свой
Coarse Raster SM



IDs of triangles that overlap tile

5) cudaraster: Coarse Raster **SM**

Подгружаем треугольники из одной и той же корзины из всех очередей (т.е. результаты всех Bin Raster **SM**s).

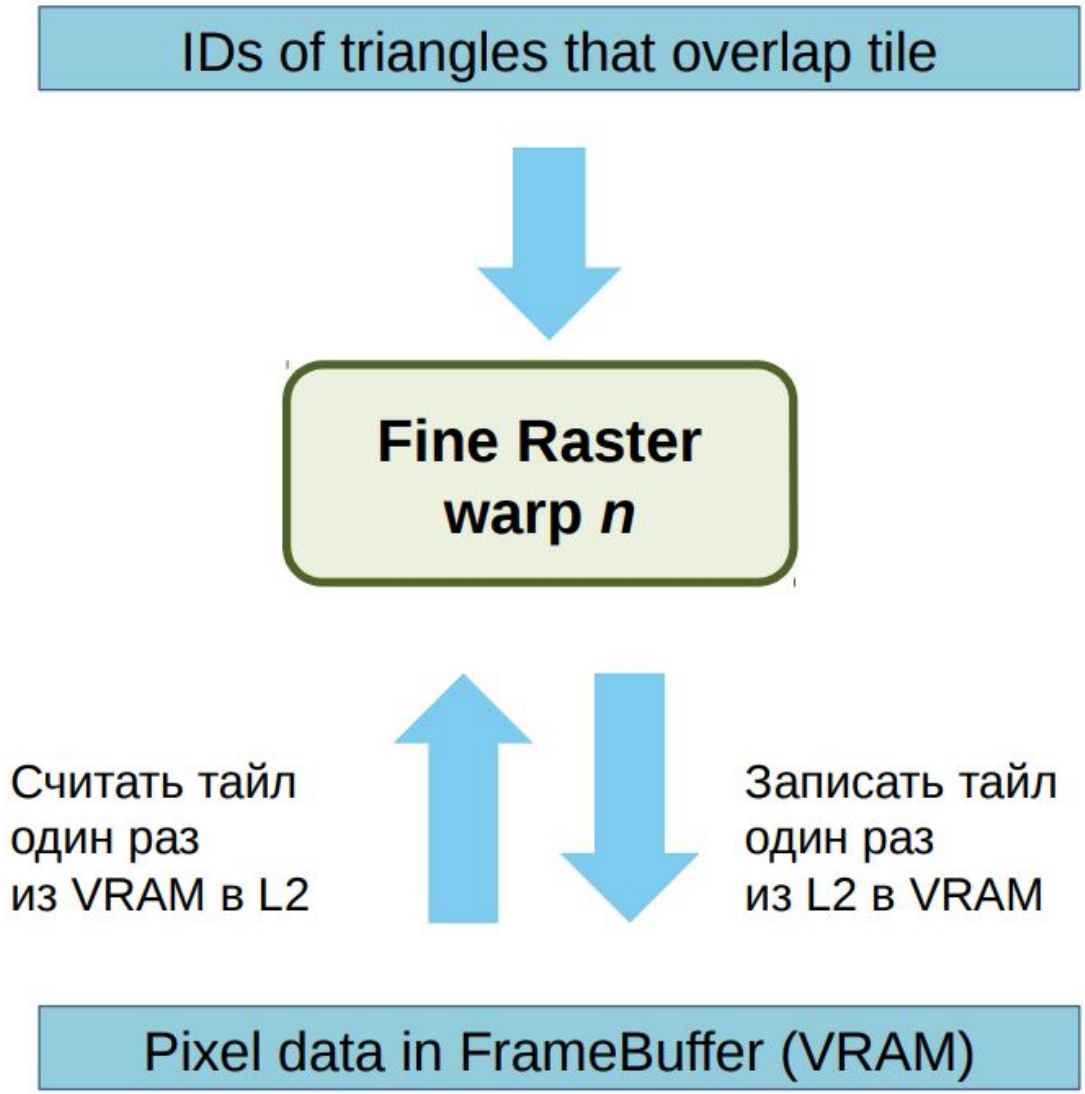
Когда треугольников подгружено достаточно для утилизации всех потоков - каждый треугольник обрабатывается своим потоком:

- Определяются **тайлы** пересекающие треугольник
- Оптимизированный код для очень маленьких и очень больших треугольников

Результат пишется в специально выделенную для данной корзины очередь (соответственно не нужна синхронизация между разными SM).

Сильно разное количество покрытых тайлов от треугольника к треугольнику (т.е. от потока к потоку). Нужно равномерно распределить запись результата.

5) cudaraster: Fine Raster **SM**



5) cudaraster: Fine Raster **SM**

Работаем с тайлом в **local memory** (i.e. **shared memory**/**L2**).

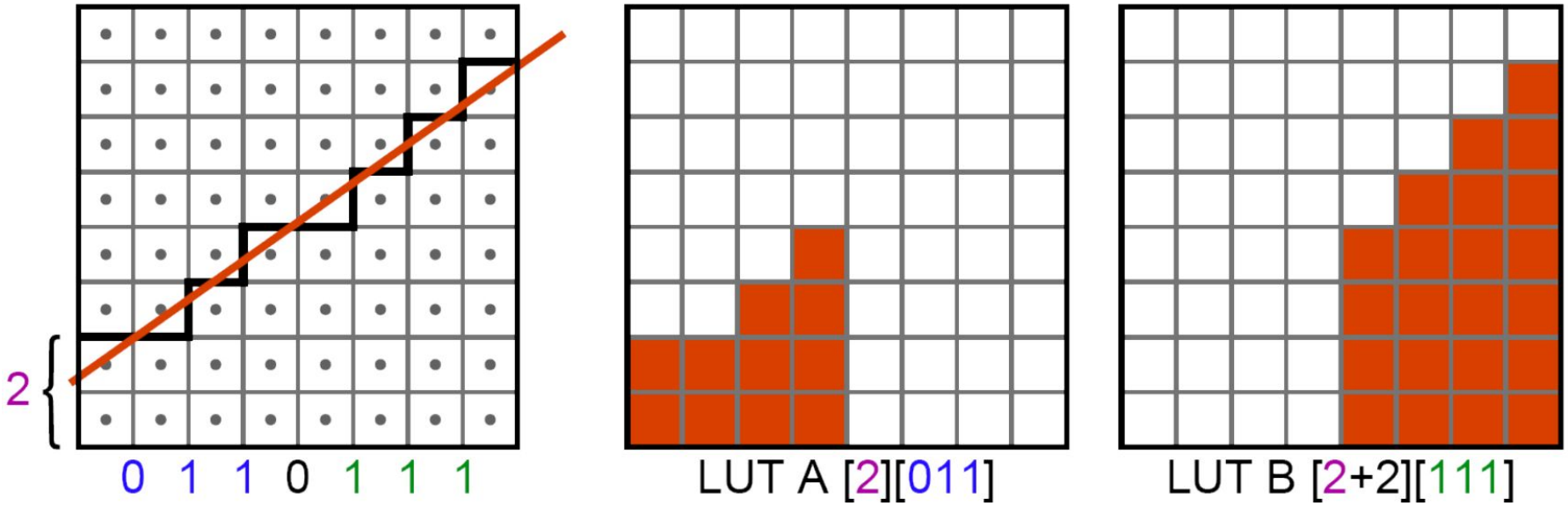
Input phase:

- Подгружаем в **L2** 32 треугольника пересекающих обрабатываемый тайл
- Откидываем треугольники сравнивая их **zmin** с **zmax** у тайла
- Рассчитываем pixel coverage используя **L**ook **U**p **T**ables: 8x8, 153 инструкции
- Повторяем пока не наберется хотя бы 32 фрагмента

Когда набралось достаточно фрагментов (или кончились треугольники),
Shading phase:

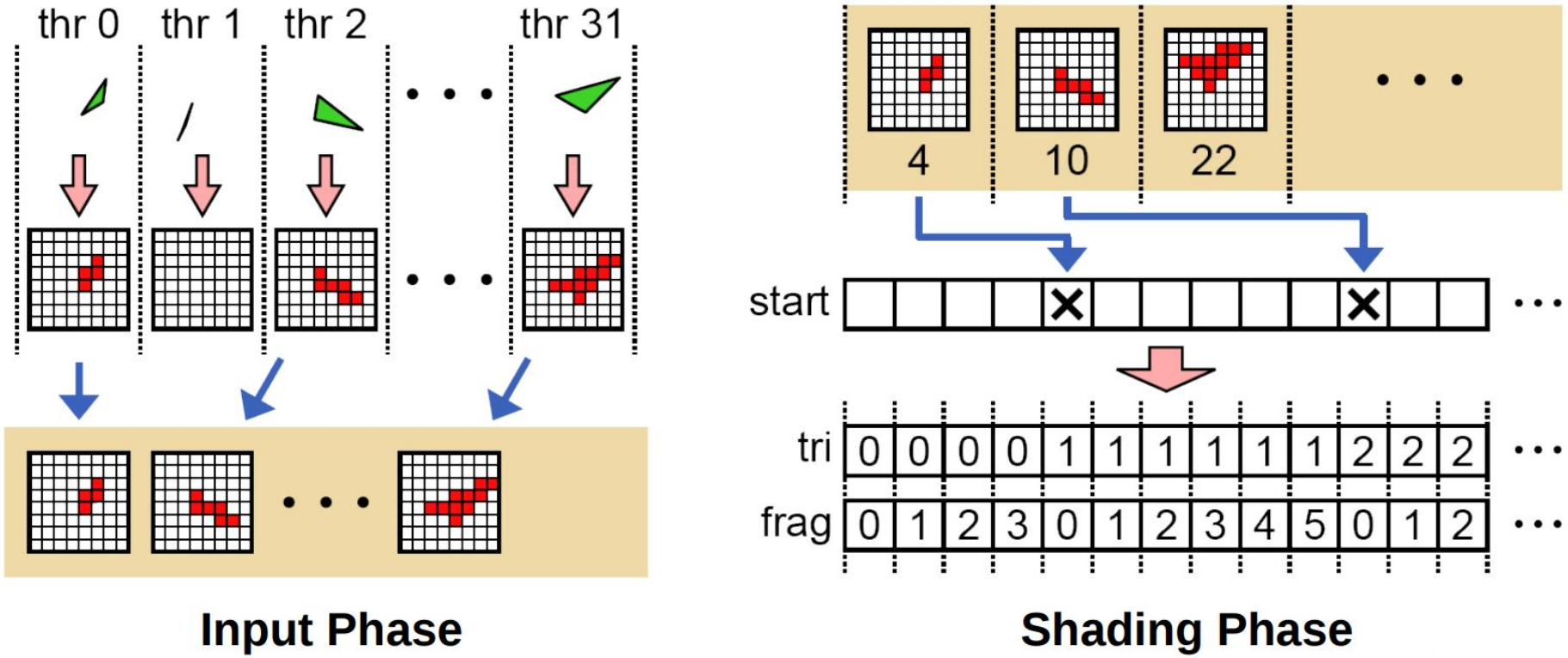
- Один фрагмент - один поток

5) cudaraster: Pixel Coverage **LUTs**



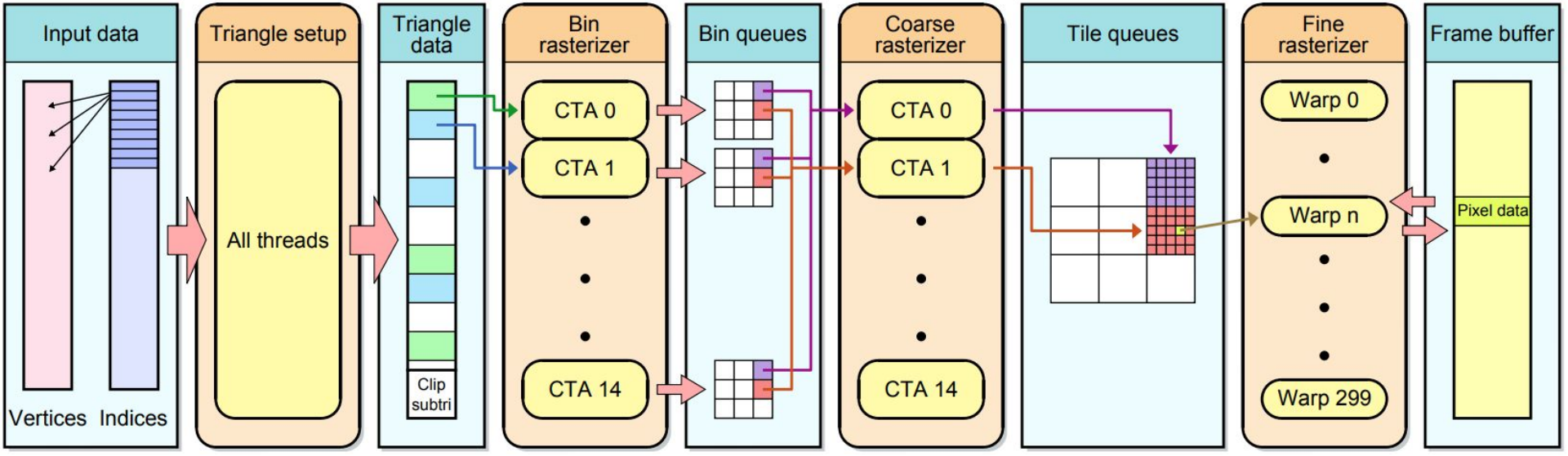
- Идем по ребру (как в алгоритме Брезенхэма)
- Используем coverage masks (маски покрытия) из **LUTs** (6 KB => L2)
- 51 инструкция на ребро (8x8)

5) cudaraster: распределение фрагментов по SM

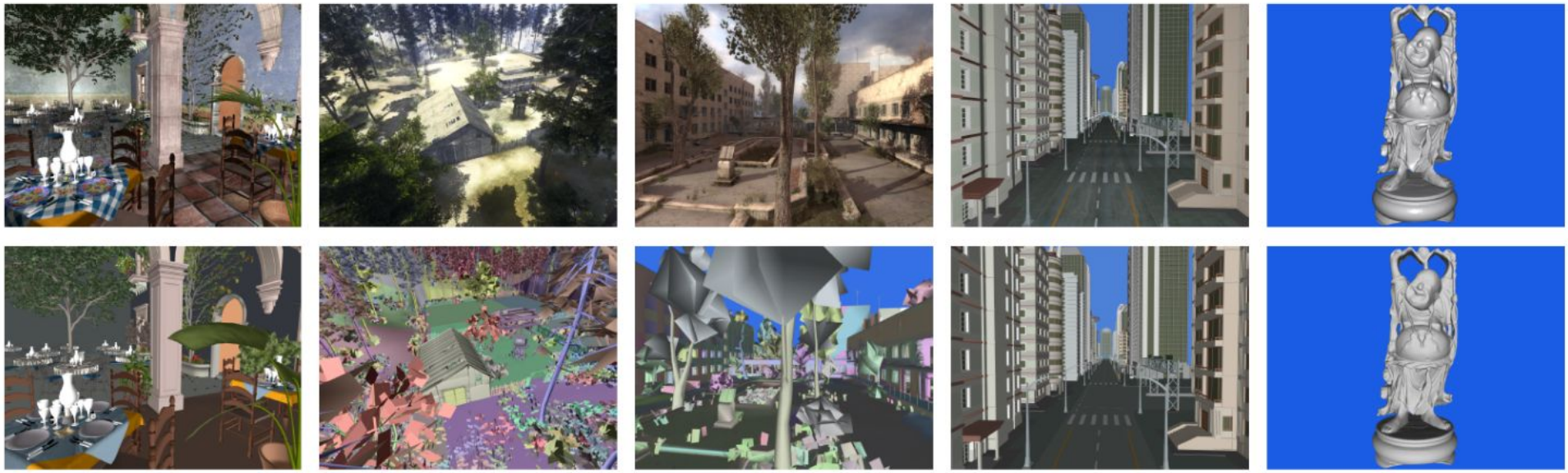


- В первой фазе рассчитываем покрытые и пишем в список
- Во второй фазе рассчитываем префиксными суммами индекс фрагмента треугольника среди все фрагментов всех обрабатываемых в данный момент треугольников

5) cudaraster



5) cudaraster: Tests



SAN MIGUEL, 189MB	JUAREZ, 24MB	STALKER, 11MB	CITY, 51MB	BUDDHA, 29MB
5.44M tris, 25% visible	546K tris, 37% visible	349K tris, 41% visible	879K tris, 21% visible	1.09M tris, 32% visible
2.4 pixels / triangle	14.6 pixels / triangle	14.1 pixels / triangle	16.3 pixels / triangle	1.4 pixels / triangle

Call of Juarez scene courtesy of **Techland**
S.T.A.L.K.E.R.: Call of Pripyat scene courtesy of **GSC Game World**

5) cudaraster: Tests

Scene	Resolution	HW	Our (SW)	FreePipe (FP)	SW:HW ratio	FP:SW ratio
SAN MIGUEL	512×384	5.37	7.82	130.14	1.46	16.65
	1024×768	5.43	9.48	510.20	1.74	53.84
	2048×1536	5.86	15.44	1652.52	2.64	107.06
JUAREZ	512×384	0.59	2.71	5.34	4.56	1.97
	1024×768	0.67	3.28	18.63	4.87	5.69
	2048×1536	1.03	7.06	72.45	6.84	10.26
STALKER	512×384	0.31	1.81	23.47	5.91	12.96
	1024×768	0.39	2.31	92.73	5.96	40.14
	2048×1536	0.67	5.41	386.07	8.10	71.36
CITY	512×384	0.93	2.16	64.56	2.32	29.88
	1024×768	1.04	3.13	251.86	3.01	80.54
	2048×1536	1.42	6.79	1032.83	4.77	152.13
BUDDHA	512×384	1.06	2.09	2.14	1.98	1.02
	1024×768	1.07	2.66	3.08	2.50	1.16
	2048×1536	1.11	4.01	6.96	3.62	1.73

Frame rendering time in **ms** (depth test + color, no MSAA, no blending)

Ссылки

- An intro to modern OpenGL. Chapter 1: The Graphics Pipeline
- habrahabr - пишем упрощённый OpenGL своими руками
- Dr.Dobb’s: Rasterization on Larrabee
- Larrabee: A Many-Core x86 Architecture for Visual Computing
- Why didn't Larrabee fail?
- High-Performance Software Rasterization on GPUs (cudaraster, paper, presentation, source code)