CS148: Introduction to Computer Graphics and Imaging

Programmable Graphics Pipelines

Topics

The fixed-function graphics pipeline

Programmable stages

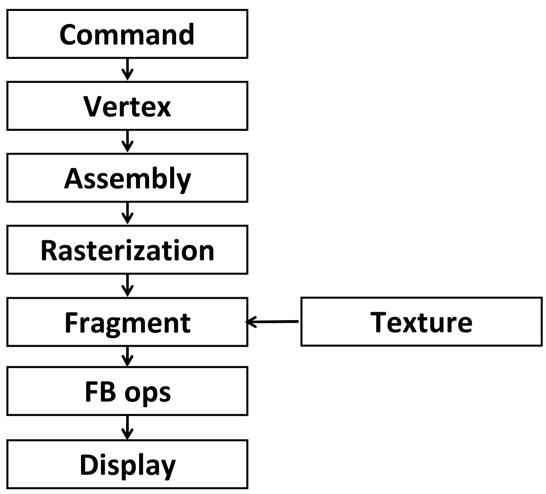
- Vertex shaders
- Fragment shaders

GL shading language (GLSL)

Mapping other applications to GPUs

The Graphics Pipeline

A Trip Down the Graphics Pipeline



Application

Simulation

Input event handlers

Modify data structures

Database traversal

Primitive generation

Graphics library utility functions (glu*)

Command

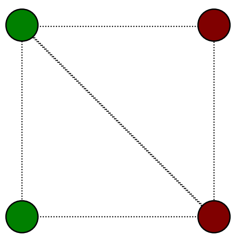
Command queue

Command interpretation

Unpack and perform format conversion

Maintain graphics state

```
glLoadIdentity( );
glMultMatrix( T );
glBegin( GL_TRIANGLE_STRIP );
glColor3f ( 0.0, 0.5, 0.0 );
glVertex3f( 0.0, 0.0, 0.0 );
glColor3f ( 0.5, 0.0, 0.0 );
glVertex3f( 1.0, 0.0, 0.0 );
glColor3f ( 0.0, 0.5, 0.0 );
glVertex3f( 0.0, 1.0, 0.0 );
glColor3f ( 0.5, 0.0, 0.0 );
glVertex3f( 1.0, 1.0, 0.0 );
…
glEnd( );
```



Vertex (per-vertex)

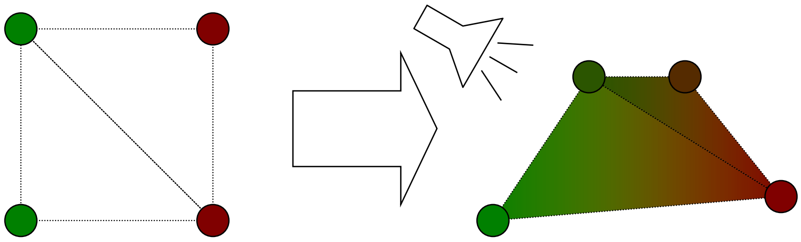
Vertex transformation

Normal transformation

Texture coordinate generation

Texture coordinate transformation

Lighting (light sources and surface reflection)



Object-space triangles

Screen-space lit triangles

Primitive Assembly

Combine transformed/lit vertices into primitives

- 1 vert -> point
- 2 verts -> line
- 3 verts -> triangle

Clipping

Perspective projection

Transform to window coordinates (viewport)

Determine orientation (CW/CCW)

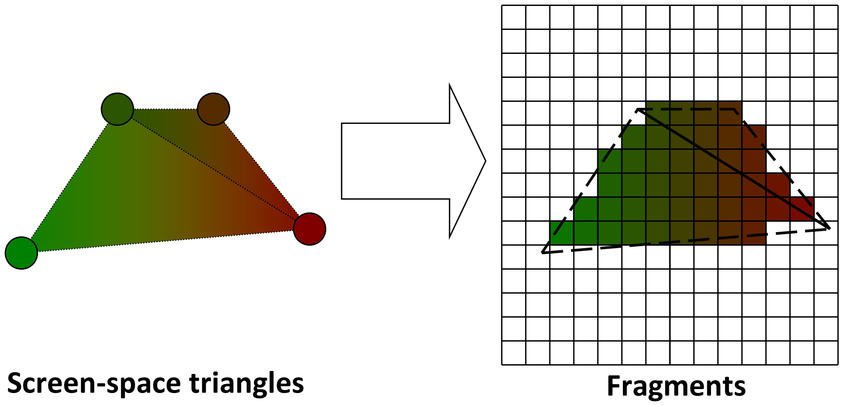
Back-face cull

Rasterization

Setup (per-triangle)

Sampling (triangle = {fragments})

Interpolation (interpolate colors and coordinates)

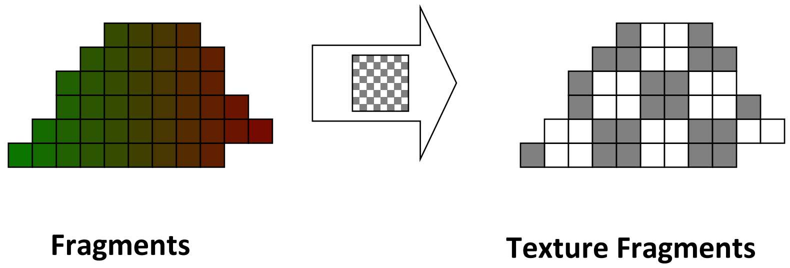


Texture

Textures are arrays indexed by floats (Sampler)

Texture address calculation

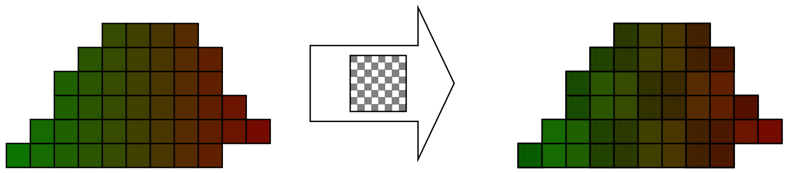
Texture interpolation and filtering



Fragment

Combine texture sampler outputs

Per-fragment shading



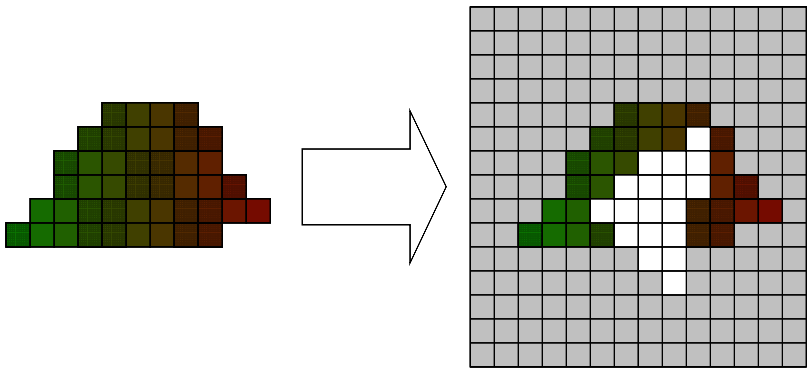
Fragments

Textured Fragments

Framebuffer Operations

Owner, scissor, depth, alpha and stencil tests

Blending or compositing



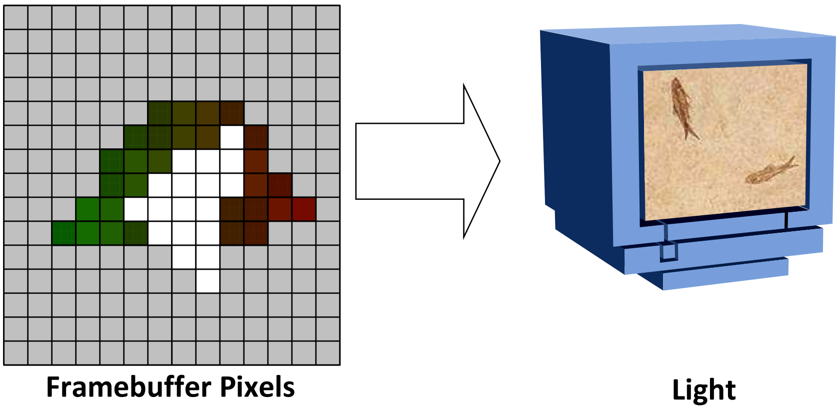
Textured Fragments

Framebuffer Pixels

Display

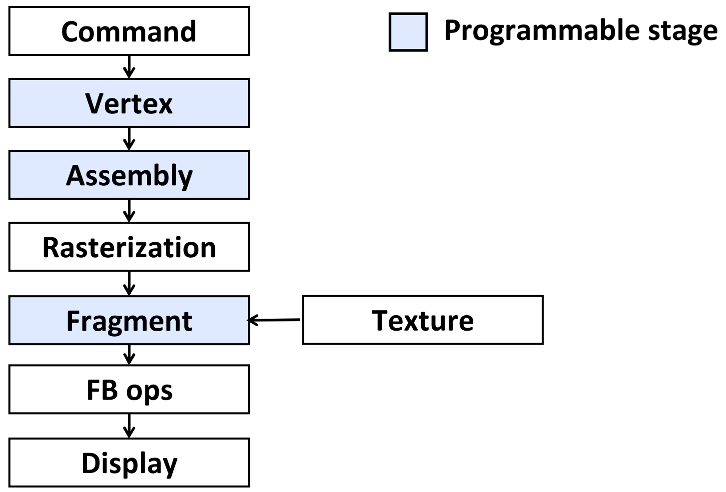
Gamma correction

Analog to digital conversion

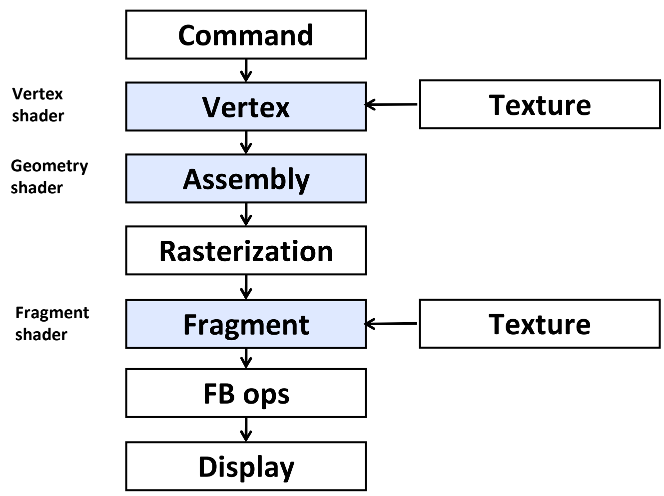


Programming Stages

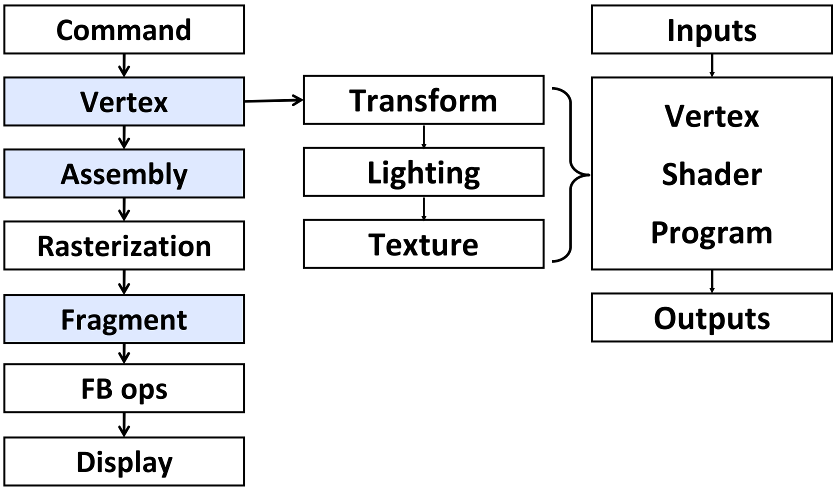
Programmable Graphics Pipeline



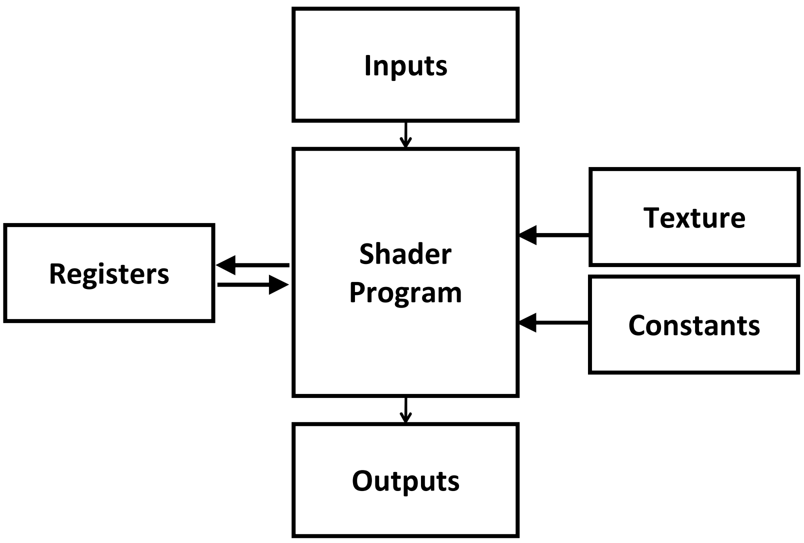
Programmable Graphics Pipeline



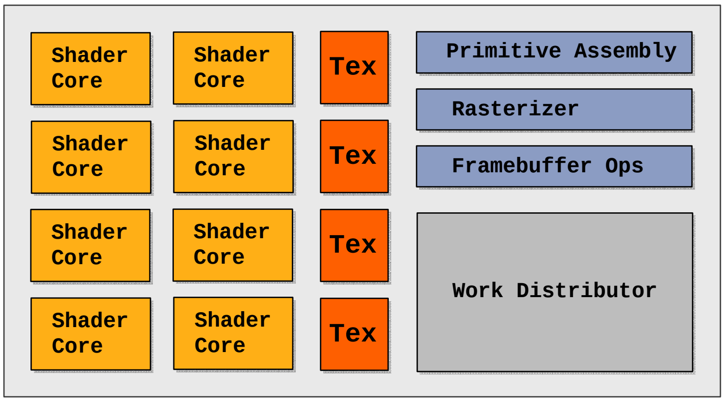
Programmable Graphics Pipeline



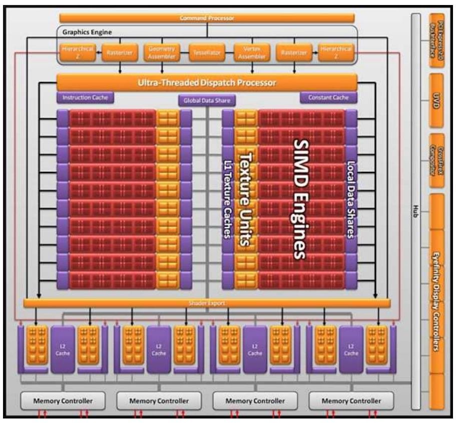
Shader Program Architecture



What's in a GPU?



What's in a GPU?



NVIDIA GF100

(GeForce GTX 480)

AMD Cypress

(Radeon HD 5870)

GLSL

Simple Vertex and Fragment Shaders

```
// simple.vert
void main()
{
    gl_Position =
      gl_ModelViewMatrix *
        gl_ProjectionMatrix * gl_Vertex;
    gl_Normal = gl_NormalMatrix * gl_Normal;
    gl_FrontColor = gl_Color;
    gl_BackColor = gl_Color;
}
// simple.frag
void main()
{
    gl_FragColor = gl_Color
}
```

Uniform Variables

Uniforms are variables set by the program that can be changed at runtime, but are constant across each execution of the shader;
Changed at most once per primitive

```
// Predefined OpenGL state
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_NormalMatrix;

// User-defined
uniform float time;
```


Attribute Variables

Attributes variables are properties of a vertex
They are the inputs of the vertex shader

```
attribute vec4 gl_Color;
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;

void main() {
    gl_FrontColor = gl_Color;
}
```

N. B. All `glVertex*()` calls result in a vec4

Varying Variables

Varying variables are the outputs of the vertex shader

```
attribute vec4 gl_Color;
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;

void main() {
   gl_FrontColor = gl_Color;
}
```


Varying Variables

The varying variables are interpolated across the triangle

`gl_Color` is set to `gl_FrontColor` or `gl_BackColor` depending on whether the triangle is front facing or back facing

```
varying vec4 gl_Color;
vec4 gl_FragColor;

void main() {
   gl_FragColor = gl_Color;
}
```

Vectors

Constructors

```
vec3 V3 = vec3(1.0, 2.0, 3.0);
vec4 V4 = vec4(V3, 4.0);
```

Swizzling

```
vec2 V2 = V4.xy;
vec4 V4Reverse = V4.wzyx;
vec4 Result = V4.xyzw + V4.xxxx;
```

Basic Vector Operators

```
float Result = dot(V4, V4Reverse);
vec3 Result = cross(V3, vec3(1.0, 0.0, 0.0));
```

N. B. Points, vectors, normals and colors are all `vec`'s

Textures

```
uniform sampler2D SomeTexture;

void main()
{
    vec4 SomeTextureColor =
        texture2D(SomeTexture, vec2(0.5, 0.5));
}
```

N. B. Textures coordinates are from (0, 0) to (1, 1)

Communicating with GLSL

Graphics state is available as uniform variables

```
uniform mat4 gl_ModelViewMatrix;
```

Can extend state

```
uniform float x;
addr = GetUniformLocation( program, "x"):
glUniform1f( addr, value );
```

Primitive attributes are available as attribute variables

Can extend attributes (inside `glBegin/glEnd`)

```
uniform float y;
addr = GetAttributeLocation( program, "y");
glVertexAttribute1f( addr, value );
```


The OpenGL Pipeline in GLSL - Vertex

Built-in attributes

```
vec4 gl_Vertex              glVertex*()
vec4 gl_Color               glColor*()
vec4 gl_SecondaryColor      glSecondaryColor*()
vec4 gl_Normal              glNormal()
vec4 gl_MultiTexCoord0      glMultiTexCoord(0, …)
```

The OpenGL Pipeline in GLSL - Fragment

Built-in varying

```
vec4 gl_Position
vec4 gl_FrontColor, gl_BackColor
vec4 gl_FrontSecondaryColor, gl_BackSecondaryColor
vec4 gl_TexCoord[n]
vec4 gl_FragCoord
```

Outputs

```
vec4 gl_FragColor
vec4 gl_FragDepth
```


Simple Pixel Shader

```
varying vec2 TexCoord0;
varying vec2 TexCoord1;
uniform sampler2D SomeTexture0;
uniform sampler2D SomeTexture1;
void main()
{
    gl_FragColor =
        texture2D(SomeTexture0, TexCoord0) * 0.5 +
        texture2D(SomeTexture1, TexCoord1) * 0.5;
}
```

This makes it easy to build image processing filters

Limitations

Memory

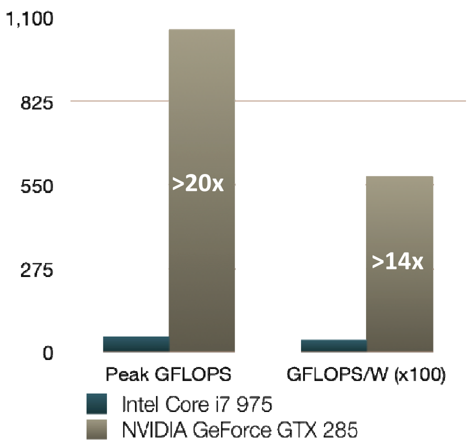
- No access to neighboring fragments
- Limited stack space, instruction count
- Cannot read and write framebuffer

Performance

- Branching support is limited and slow
- Graphics card will timeout if code takes too long
- Variable support across different graphics cards

GPU Computing

Why GPGPU?



GPU's are great if problem:

- Executes the same code many times on different input
- Needs lots of math
- Does not share data between executing components
- Has lots of work to do without CPU intervention

Computation on GPU's

Beyond basic graphics pipeline

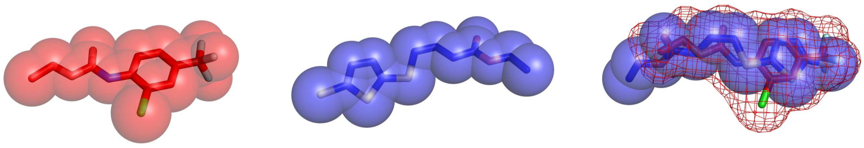
- Collision detection
- Fluid and cloth simulation
- Physics
- Ray-tracing

Beyond graphics

- Protein folding (Folding@Home)
- Speech recognition
- Partial differential equation solvers
- Fourier transforms

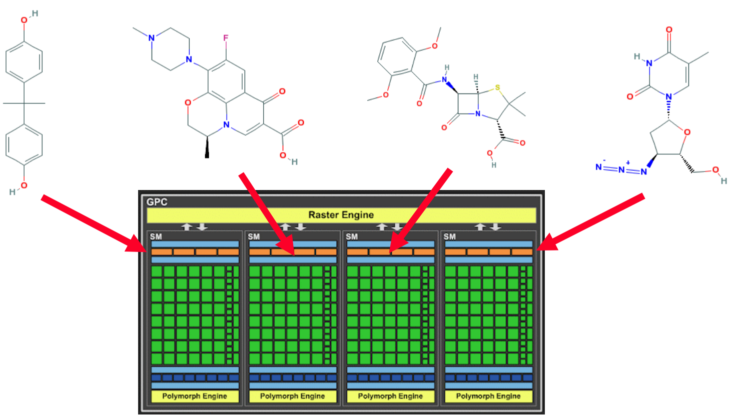
An Example GPGPU Application - PAPER

- Molecular overlay optimization: used in computational drug discovery to find new active compounds from a database given one active "query" molecule



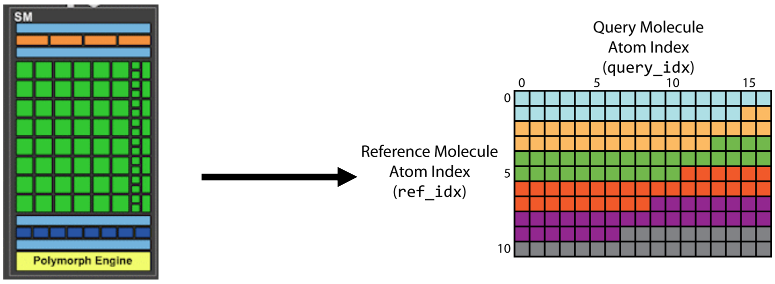
- Complexity O(MN): double-loop over all atom pairs
- DB = ~10M molecules; CPU = 10ms/overlay = ~2 days/query
- *Use GPU to exploit parallelism of problem.*

GPU Parallelism 1



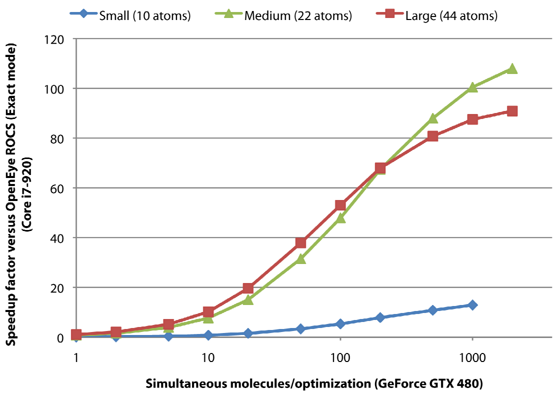
Each optimization is independent, and each SM (OpenCL work-group) executes independently, so run one DB molecule per GPU core

GPU Parallelism 2



GPU cores have wide internal parallelism. Each atom pair in an optimization is independent – map each to a shader unit (OpenCL work-item), and loop.

GPGPU Conclusion



>100x speedup if there's lots of parallel work

48 hr for CPU DB search -> 30-60 min with GPU!