Metal

GPU-accelerated advanced 3D graphics rendering and data-parallel computation
Maths

The heart and foundation of computer graphics
Metalmatics

There are no computer graphics without mathematics

Rasterized 3D graphics requires algebra

  Vector operations

  Matrix operations

  Interpolation techniques
My first triangle
Why triangles?

Easiest surface that can be formed by straight lines

Other surfaces can be composed by a sequence of triangles

http://en.wikipedia.org/wiki/Polygon_mesh
3D graphics pipeline

Draw Call → Vertex Function → Rasterization → Fragment Function → Screen
A collection of states to render our triangle

- Vertices
- Uniform data
- Vertex Function
- Fragment Function
Vertex (Vertices)

Structure that holds all the information of a triangle’s corner

For our first triangle

   Position: x, y

   Color: red, green, blue
struct Vertex {
    float x, y;
    float red, green, blue;
};

Vertex triangle[3] = {
    { -1.0, -1.0,  1.0, 0.0, 0.0 },
    {  1.0, -1.0,  0.0, 1.0, 0.0 },
    {  0.0,  1.0,  0.0, 0.0, 1.0 }
};
Coordinate System

Metal uses a right handed coordinate system

- Positive X axis points right
- Positive Y axis points up
- Positive Z axis point forward

Use your right hand for visualization

http://en.wikipedia.org/wiki/Coordinate_system
Uniform data

A structure that holds all data that is the same per draw call
For our triangle it is a rotation around the z-axis by angle $\alpha$

```cpp
struct Uniforms {
    float_matrix_4x4 transformation;
};

Uniforms uniform = {
    cos(a), sin(a), 0.0, 0.0,
    -sin(a), cos(a), 0.0, 0.0,
    0.0,     0.0,     1.0, 0.0,
    0.0,     0.0,     0.0, 1.0
};
```
Transformation matrices

All transformations for 3D math can be expressed by a $4 \times 4$ matrix easy to create and fast to apply (to each vertex).

CGAffineTransform

Representation of a $2 \times 3$ matrix for 2D projection

CGAffineTransformIdentity

CGAffineTransformMakeRotation(angle)

CGAffineTransformMakeScale(x, y)

CGAffineTransformMakeTranslation(x, y)
Demo

Yes, it is Comic Sans
Vertex Function

Handles a single vertex

Called for each vertex in the draw call

Transforms the position to the final position

Optionally passes other uniforms to the fragment function
Vertex Function

```cpp
struct Output {
    float4 position;
    float3 color;
};

Output vertexFunction(Vertex v, Uniforms u) {
    Output o;
    o.position = u.transform * v.position;
    o.color = v.color;
    return o;
}
```
Rasterization

Converts the transformed positions into fragments

A fragment is a future pixel

Interpolates the other vertex values (color)

http://en.wikipedia.org/wiki/Rasterisation
Fragment Function

Handles a single fragment

Called once per fragment

Converts the fragment into a pixel on the screen
Fragment Function

// Output of the Vertex Function without position
struct Input {
    float3 color;
};

float3 fragmentFunction(Input i) {
    float3 r;
    r = 1.0 - i.color;
    return r;
}
Homework

Write your own software rasterizer
What is Apple’s Metal API?

The Metal framework supports GPU-accelerated advanced 3D graphics rendering and data-parallel computation workloads. Metal provides a modern and streamlined API for fine-grain, low-level control of the organization, processing, and submission of graphics and computation commands and the management of the associated data and resources for these commands. A primary goal of Metal is to minimize the CPU overhead necessary for executing these GPU workloads.

– Metal Programming Guide
Apple’s Graphics APIs

Is there room left for more graphics APIs?

CoreGraphics / Quartz

CoreImage

CoreText

OpenGL / OpenGLES

SpriteKit (2D) / SceneKit (3D)
Metal features

Designed for performance

  Reduce CPU overhead while maximizing the GPU performance

3D raster graphics pipeline

Parallel computing pipeline

  Replaces OpenCL (was never public on iOS)
Metal requirements

iOS8 only – Mac support would require unified RAM

ARM64 only – iPhone 5s, iPad Air, iPad mini with Retina Screen

Device only – no simulator support

Xcode 6
Metal API

Quite an unusual API for an Apple framework

1× Function

23× Classes (all helpers)

18× Protocols (core)

a few enumerations and type definitions
First step is to get the GPU

```swift
id<MTLDevice> device = MTLCreateSystemDefaultDevice();
```
Metal will draw to a layer, there is no UIMetalView

```swift
#import <QuartzCore/CAMetalLayer.h>

CAMetalLayer *layer = [CAMetalLayer layer];
layer.device = device;
layer.pixelFormat = MTLPixelFormatBGRA8Unorm;
layer.frame = self.view.layer.bounds;
[self.view.layer addSublayer:layer];
```
CAMetalDrawable

Request a drawable to render into

Might return nil when rendering too fast

$id<$CAMetalDrawable$> drawable = layer.nextDrawable;
MTLBuffer

A buffer contains data that is shared between CPU and GPU

You are responsible for synchronized access!

```swift
id<MTLBuffer> vertexBuffer = [device
    newBufferWithBytes:vertices
    length:sizeof(vertices)
    options:MTLResourceOptionCPUCacheModeDefault];

id<MTLBuffer> uniformBuffer = [device
    newBufferWithBytes:uniforms
    length:sizeof(uniforms)
    options:MTLResourceOptionCPUCacheModeDefault];
```
MTLLibrary

A library is containing all the shaders (vertex and fragment functions)

There might be multiple libraries

id<MTLLibrary> library = [device newDefaultLibrary];
MTLFunction

MTLFunction (shader) is code that is executed on the GPU.

```objectivec
id<MTLFunction> vertexFunction = [library newFunctionWithName:@"vertexFunction"];  

id<MTLFunction> fragmentFunction = [library newFunctionWithName:@"fragmentFunction"];
```
MTLCommandQueue

Command Queues execute Command Buffers in serial order

```swift
id<MTLCommandQueue> commandQueue = [device newCommandQueue];
```
MTLRenderPassDescriptor

Describes the buffer use to render to

MTLRenderPassDescriptor *renderPass =  
    [MTLRenderPassDescriptor renderPassDescriptor];

renderPass.colorAttachments[0].texture = drawable.texture;  
renderPass.colorAttachments[0].loadAction =  
    MTLLoadActionClear;  
renderPass.colorAttachments[0].clearColor =  
    MTLClearColorMake(1.0f, 1.0f, 1.0f, 1.0f);  
renderPass.colorAttachments[0].storeAction =  
    MTLStoreActionStore;
MTLRenderPipelineDescriptor

Describes the pipeline that is used for rendering

Should be prepared in the setup phase

MTLRenderPipelineDescriptor *pipelineDesc =
    [[MTLRenderPipelineDescriptor alloc] init];

[pipelineDesc setVertexFunction:vertexFunction];
[pipelineDesc setFragmentFunction:fragmentFunction];

id<MTLRenderPipelineState> pipeline = [device
    newRenderPipelineStateWithDescriptor:descriptor error:nil];
MTLCommandBuffer

Command Buffers hold series of Command Encoders

```swift
id<MTLCommandBuffer> commandBuffer = [commandQueue commandBuffer];

// fill with Command Encoders (next slide)
[commandBuffer addCompletedHandler:...];
[commandBuffer presentDrawable:drawable];
[commandBuffer commit];
```
MTLCommandEncoder

Command Encoders hold a sequence of GPU instructions

```swift
id<MTLRenderCommandEncoder> commandEncoder = [commandBuffer renderCommandEncoderWithDescriptor:renderPass];

[commandEncoder setRenderPipelineState:pipeline];
[commandEncoder setVertexBuffer:vertices offset:0 atIndex:0];
[commandEncoder setVertexBuffer:uniforms offset:0 atIndex:1];

[commandEncoder drawPrimitives:MTLPrimitiveTypeTriangle vertexStart:0 vertexCount:3 instanceCount:1];

[commandEncoder endEncoding];
```
Metal Shading Language

The shading language is C++11

Custom standard library

Math libraries

Additional keywords
Restrictions

Some C++11 features are missing

new and delete operator

Exceptions

Lambdas

Exceptions

virtual functions

...
Benefits

Shared code between CPU and GPU
  Helpful for vertex and uniforms structs
Compiled with the clang toolchain
  Precompiled at compile time
Shared types

typedef struct {
    matrix_float4x4 transform;
} Uniforms;

typedef struct {
    packed_float2 position;
    packed_float3 color;
} Vertex;

typedef struct {
    float4 position [[ position ]];
    half4 color;
} Varyings;
Vertex Function

vertex Varyings vertexFunction(
    device Vertex* vertex_array [[ buffer(0) ]],
    constant Uniforms& uniforms [[ buffer(1) ]],
    unsigned int vid [[ vertex_id ]]
) {
    Varyings out;
    Vertex v = vertex_array[vid];

    float4 position = float4(v.position, 0.0, 1.0);
    out.position = uniforms.transform * position;
    out.color = half4(half3(v.color), 1.0);

    return out;
}
Fragment Function

```cpp
fragment half4 fragmentFunction(
    Varyings in [[ stage_in ]]
) {
    return 1.0 - in.color;
}
```
Kernel Function

```c
kernel void kernelFunction(
    const device float *in0 [[ buffer(0) ]],
    const device float *in1 [[ buffer(1) ]],
    device float *out [[ buffer(2) ]],
    uint id [[ thread_position_in_grid ]]
) {
    out[id] = in0[id] + in1[id];
}
```
vertex Varyings vertexFunction(
  device Vertex* vertex_array [[ buffer(0) ]],
  constant Uniforms& uniforms [[ buffer(1) ]],
  unsigned int vid [[ vertex_id ]]
)
{
  Varyings out;
  Vertex v = vertex_array[vid];
  float4 position = float4(v.position, 0.0, 1.0);
  out.position = uniforms.transform * position;
  out.color = half4(half3(v.color), 1.0);
  return out;
}

fragment half4 fragmentFunction(
  Varyings in [[ stage_in ]]
)
{
  return 1.0 - in.rgb;
}
GPU Frame Debugger

Inspect the state of the pipeline

Live edit shaders

View buffers and textures

Also works for OpenGL ES on iOS
/**
 * Copyright (C) 2014 Apple Inc. All Rights Reserved.
 * See LICENSE.txt for this sample's licensing information
 *
 * Abstract:
 * View for Deferred lighting Metal Sample Code. Manages framebuffers and expects a delegate to respond to render commands to perform drawing. Can be configured with 4 color attachments, depth and stencil attachments.
 *
 */

#import "AAPLView.h"

@implementation AAPLView
{
@private
    __weak CAMetalLayer * _metalLayer;
    BOOL _layerSizeDidUpdate;
    id <MTLTexture> _depthTex;
    id <MTLTexture> _stencilTex;
    id <MTLTexture> _colorTextures[3]; // these are for textures 1-3 (as needed), texture 0 is owned by the drawable

    @synthesize currentDrawable = _currentDrawable;
    @synthesize renderPassDescriptor = _renderPassDescriptor;
    ... (rest of the implementation...)

Location: 82272, length: 48
2014-08-28 09:03:16.976
Demo
Metal vs OpenGL
OpenGL Problems

State validation ✓ done once at setup
Buffer copies ✓ no copies, but programmer is responsible for synchronizing
Runtime shader compilation ✓ can be done at compile time
Black screen ✗ random effects
When to use Metal

Indirectly when using a 3D engine

Unity, Unreal, (SpriteKit, SceneKit)

Custom high-performance 3D engine

Parallel computation

No OpenCL, only alternative is misusing OpenGL