

Графический API Apple- устройств - теперь и на C++

История/особенности/микро введение в Metal

План доклада

Немного  магии

- История трехмерной графики на Apple устройствах
- Зачем нужен Metal. Есть ли смысл его использовать?
- Как программировать на metal. Можно ли сразу использовать C++?
- Metal-cpp - что за библиотека и зачем она нужна
- MSL - язык шейдеров - это обычный C++? Переход от HLSL к MSL

О чем мы не поговорим

А жаль

- О том как программировать трехмерную графику
- О практическом применении Metal

Стародавние времена

Сначала впереди, а в конце позади

- Одна из первых по настоящему стабильных OpenGL имплементаций
- Зависимость и постоянные разборки с NVIDIA/AMD
- Они же писали драйвера
- LLVM родился для OpenGL
- OpenGL 4.1+ и на этом все
- Никакого инструментария/никакой отладки
- Заморожен и переписан с использованием Metal для ARM64

Современные реалии

Становится веселее



- Metal был первым на празднике жизни нового поколения GPU API
- Metal изначально заточен на свои собственные GPU
- Metal используется во всех устройствах Apple
- API для разных устройств имеет минимальные отличия
- iOS/iPad OS/macOS/tvOS могут использовать одно и то же API
- Хороший инструментарий

Знакомство с Metal

И даже на Intel архитектуре

- Apple M1/M2/M3 CPU
- Дискретные устройства AMD в ноутбуках
- Интегрированная графика Intel
- Полноценные AMD видеокарты в MacPro
- Любое другое устройство - даже Apple Watch
- Симулятор на Mx CPU

Знакомство с Metal

Но лучше на ARM64

- Не такие простые видеочипы M1/M2/...
- TBDR архитектура
- Общая память, и ее может быть очень много
- RTX
- Специфические фишки платформы (tile shaders, programmable blending)
- VScx компрессия из коробки
- И много много другого...

Зачем нужен Metal

Есть же прекрасные кроссплатформенные технологии ...

- Удобно/быстро/поддержка разнообразных утилит
- Графика это 5% кода, а драма обычно на все 50%
- Использование ресурсов GPU для вычислений
- Поддержка всего спектра устройств Apple
- Отладка шейдеров
- Валидация команд

В чем суть Metal

Параллельная вселенная

- CPU/GPU должны работать параллельно, каждый в своей вселенной, предпочтительно без синхронизации
- API простое и по умолчанию все делает за разработчика
- При желании Metal может перейти в полностью ручной режим
- Metal полностью многопоточная система, разработчик может загружать GPU параллельно из нескольких потоков
- API ориентированно на будущее - с прицелом на GPU Driven Pipelines

Как программировать Metal

На сцену выходит Objective-C

- Основной язык Apple API - Objective C - бодрый старик с богатой историей
- Swift вне нашего интереса
- Полностью интегрирован в современные CLANG компиляторы C++
- Objective C++ (.m и .mm файлы)
- Идет с ОС фреймворками Cocoa и имеет два режима управления памятью
- Написать программу для Metal полностью на C++ без Obj-C было НЕВОЗМОЖНО

Как программировать Metal

А так ли все грустно?

- При классическом разделении графического API на отдельную подсистему задача решается тривиально
- C++ классы могут вызывать методы ObjC без каких либо ограничений
- Синтаксический ‘суп’ как наиболее неприятное последствие
- C++ классы могут хранить указатели на ObjC классы

- ```
// Draw
void xx::metal::MetalRenderer::DrawArrays(const int firstVertex, const int nVertices)
{
 PreDraw();

 [m_RenderCommandEncoder drawPrimitives: xx2MTL::PrimitiveType(m_CurrentState.m_PrimitiveTopology) vertexStart:firstVertex vertexCount:nVertices];

 PostDraw();
}
```

# Как программировать Metal

## А так ли все грустно?

- А что такое m\_RenderCommandEncoder?

```
OBJC_ID(MTLRenderCommandEncoder) m_RenderCommandEncoder;
```

```
#ifdef __OBJC__
include <Metal/Metal.h>
define OBJC_I(type) type*
define OBJC_ID(type) id<type>
#else
define OBJC_I(type) void*
define OBJC_ID(type) void*
#endif
```

- Единственное тонкое место - управление памятью

# Как программировать Metal

## На сцену выходит metal-cpp

- <https://developer.apple.com/metal/cpp/>
- Вращатель над ObjC вызовами/вызовы сделаны на самом низком уровне - почти нулевая потеря производительности
- Полная имплементация всех API, без каких либо расхождений для всех устройств
- Возможность компоновки всего в единственных header файл

```
#define NS_PRIVATE_IMPLEMENTATION
#define CA_PRIVATE_IMPLEMENTATION
#define MTL_PRIVATE_IMPLEMENTATION
#include <Foundation/Foundation.hpp>
#include <Metal/Metal.hpp>
#include <QuartzCore/QuartzCore.hpp>
```

# СРАВНИМ ВЫЗОВЫ

## C++ vs ObjC++



```
MTL::CommandBuffer* pCmd = _pCommandQueue->commandBuffer();
MTL::RenderCommandEncoder* pEnc = pCmd->renderCommandEncoder(pRpd);

pEnc->setRenderPipelineState(_pPSO);
pEnc->drawPrimitives(MTL::PrimitiveTypeTriangle,
 NS::UInteger(0),
 NS::UInteger(3));

pEnc->endEncoding();
pCmd->presentDrawable(pView->currentDrawable());
pCmd->commit();
```

metal-cpp

Objective-C

```
id<MTLCommandBuffer> cmd = [_commandQueue commandBuffer];
id<MTLRenderCommandEncoder> enc = [cmd renderCommandEncoderWithDescriptor:pRpd];

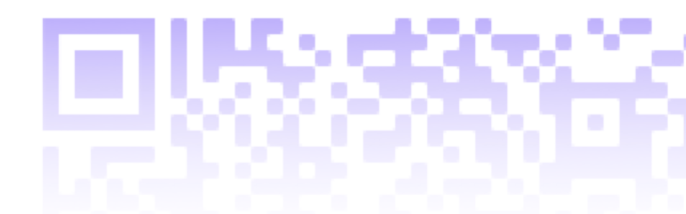
[enc setRenderPipelineState:_pPSO];
[enc drawPrimitives:MTLPrimitiveTypeTriangle
 vertexStart:0
 vertexCount:3];

[enc endEncoding];
[cmd presentDrawable:view.currentDrawable];
[cmd commit];
```

# Как программировать Metal

## Что там с памятью моей стало?

- Два режима управления - ручной и автоматический
- Все объекты Metal наследники базового класса в котором реализован подсчёт ссылок
- Для проектов полностью на C++ имеет смысл глобально переключить режим управления памятью в ручной режим
- Почти все объекты Metal удаляются отложено поэтому нужно помнить про возможность разрастания пула памяти без использования autorelease
- Обязательно читаем полезную и нужную документацию

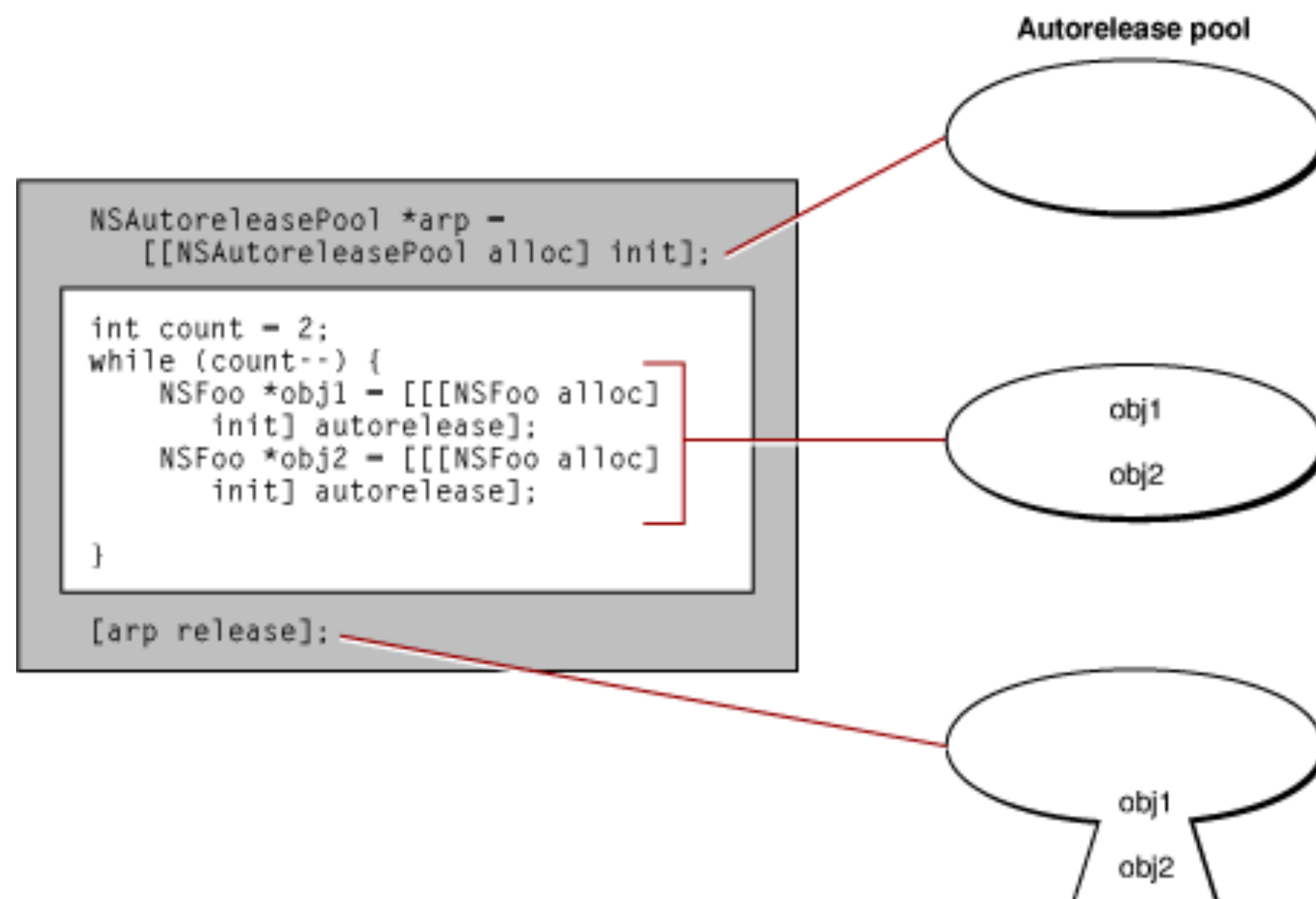


# Пронизывает все классы macOS



# Отложенные пулы

90% объектов Metal отложенные



# Как программировать Metal

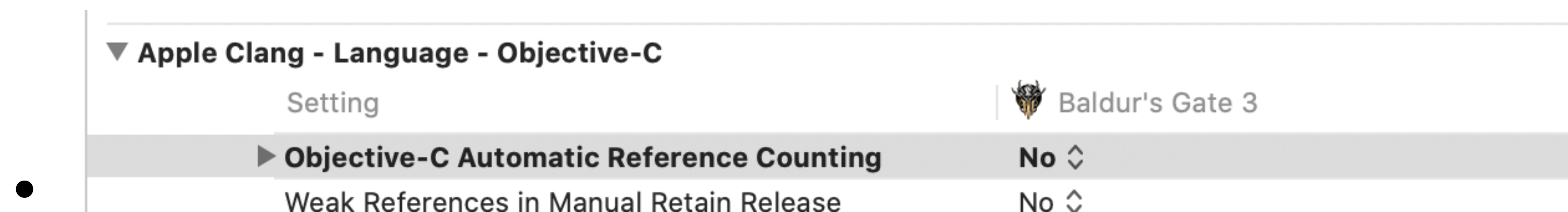
## Что там с памятью моей стало?

- Cocoa memory management обязателен для чтения и понимания
- К счастью инструменты анализа памяти на высоте
- Zombie анализ эффективно найдёт проблемы
- Другие инструменты могут визуализировать полный жизненный путь любого объекта
- Для графики нормально иметь сотни временных объектов, но ненормально постоянно аллоцировать память во время построения кадра

# Как программировать Metal

## Требуются примеры

- Самый простой пример встроен в Xcode (Game project)
- Достаточно много примеров имеют C++ вариант
- <https://developer.apple.com/metal/sample-code/>



- Наиболее простой метод учиться трехмерной графике - это включить валидатор и отлавливать кадры

# А как же шейдеры?

Они всегда были на C++

- Язык шейдеров Metal - MSL это C++
- Ограничений много, но они разумные - нет lamda, exceptions, rtti
- Нет new/delete и vtbl
- Нет goto
- Нет std
- Своя библиотека
- Стандартный препроцессор



# HLSL vs MSL

## Выглядит грустно

- Потому что нет глобальных переменных
- Потому что матрицы перевернуты
- Потому что мы формируем библиотеку при помощи компоновщика
- Потому что часть исходного кода уже потеряна
- Потому что люди ленивы

# HLSL vs MSL

## Кто что напридумывал

- GamePortingToolkit от Apple
- ShaderConductor от Microsoft
- SpirV-Cross
- Тысячи их
- SLANG (наш личный проект, доработанный напильником)
- Руки.sys никто не отменял

# Типичный HLSL

## BC4 Compress

```
uint2 CompressAlphaBlock(in const float2 iTc)
{
 int texels[16];
 ExtractAlphaBlock(texels, iTc);

 uint endpoints6[2] = {0, 0};
 uint endpoints8[2] = {0, 0};

 float error6 = 0.0f;
 float error8 = 0.0f;
 bool flat = false;

 Compute6(endpoints6, error6, flat, texels);

 if (!flat)
 Compute8(endpoints8, error8, texels);
 else
 error8 = 1000000.0f;

 uint i[16];
 uint selected_endpoints[2];
 [branch]
 if (error8 <= error6)
 {
 selected_endpoints[0] = endpoints8[0];
 selected_endpoints[1] = endpoints8[1];
 }
 else
 {
 selected_endpoints[0] = endpoints6[0];
 selected_endpoints[1] = endpoints6[1];
 }

 Encode(i, selected_endpoints, flat, texels);

 return block;
}
```

```
PsOut main(in const PsIn psIn)
{
 PsOut psOut;
 psOut.block = CompressAlphaBlock(psIn.texCoord);
 return psOut;
}
```

# Типичный MSL

## BC4 Compress

```
static uint2 CompressAlphaBlock(float2 iTc, sampler PointClampSampler, texturecube<float>
source, constant const _CompressParams& CompressParams)
{
```

```
 int texels[16];
 ExtractAlphaBlock(texels, iTc, PointClampSampler, source, CompressParams);
```

```
 uint endpoints6[2];
 uint endpoints8[2];
```

```
 float error6 = 0.0f;
 float error8 = 0.0f;
 bool flat = false;
```

```
 Compute6(endpoints6, error6, flat, texels);
```

```
 if (!flat)
 Compute8(endpoints8, error8, texels);
```

```
 else
 error8 = 1000000.0f;
```

```
 uint i[16];
 uint selected_endpoints[2];
 //[branch]
```

```
 if (error8 <= error6)
 {
 selected_endpoints[0] = endpoints8[0];
 selected_endpoints[1] = endpoints8[1];
 }
```

```
 else
 {
 selected_endpoints[0] = endpoints6[0];
 selected_endpoints[1] = endpoints6[1];
 }
```

```
 Encode(i, selected_endpoints, flat, texels);
```

```
 return block;
```

```
}
```

```
fragment uint2 CompressBC4_PS(PsIn input [[stage_in]], sampler
PointClampSampler, texturecube<float> source, constant const _CompressParams&
CompressParams)
{
```

```
 uint2 Out;
 Out = CompressAlphaBlock(input.texCoord, PointClampSampler, source,
CompressParams);
```

```
 return Out;
}
```

# HLSL vs MSL

## Практические советы

- Основная головная боль - это глобальные переменные, все остальное ерунда
- Решений несколько:
  - Передавать глобальные переменные как аргументы функций
  - Создать структуру обертку и передавать только ее
  - Создать класс обертку и работать в его рамках

# HLSL vs MSL

## Практические советы

- MSL пытается жестко оптимизировать, но если ему не помогать..
- Передаем все по ссылке, в HLSL вообще такого понятия нет там все всегда по ссылке
- Не увлекаемся вложенными структурами (особенно пустыми)
- Помним о том что большинство типов (float3) выровнены и поинтерная математика на них приведет к неверным результатам (packed\_float3)
- Активно используем air-objdump с ключом -disassemble-all чтобы посмотреть на получившуюся IR
- Наличие metcpu() - верный признак что все пошло не так

# HLSL vs MSL

## Практические советы

- Минимизируем über шейдера
- Не пытаемся делать inline за компилятор или разворачивать циклы
- static const превращается в constant const
- Можно пользоваться шаблонами особенно это важно при поддержке half/float вариантов
- Собираем все через консоль - так оно проще и нагляднее
- xcrun metal, metallib, metal-ar

# ПОДЫТОЖИМ

## Что нового мы узнали сегодня

- Objective-C легко превращается в Objective-C++ и при правильном формировании проекта нам не нужно никаких дополнительных библиотек
- При категорическом его неприятии есть metal-cpp
- В любом случае нужно знать как управлять памятью в Cocoa программе
- Программировать metal просто - не надо мучаться с кросс решениями - надо просто попробовать
- Перевод HLSL->MSL неприятная задача и лучше ее решать приятными подходами, чтобы не было мучительно больно потом

# Подытожим

## Что нового мы узнали сегодня



- На сайте Apple полно примеров и хорошая документация
- Мх устройства предпочтительная платформа для разработки
- Инструментарий более чем адекватный
- Большинство технологий и подходов работают на всех устройствах
- Apple GamePortingToolkit интересен для опытных разработчиков на DX12



# Вопросы