



The 21th
Visualization
Conference

【OpenGLジャパン】

次世代グラフィックスAPI Vulkan/ OpenGL/OpenGL ES最新情報

株式会社デジタルメディアプロフェッショナル
常務取締役開発統括部長 大淵 栄作

デジタルメディアプロフェッショナル概要

- 設立: 2002年7月
- 所在地: 東京都中野区
- 資本金: 8億2千万円
- 主要株主: UKCホールディングス、他
- 事業内容: 1. グラフィックスIPコアのライセンス
2. グラフィックスLSIのファブレス開発・製造



当社IPコアを使った顧客製品例



本社(中野セントラルパーク)



当社LSI製品

- 特許等: 日米欧において63件の特許申請済み 成立38件

今日のトピックス

- Khronos API 最新アップデート
- Vulkan - 次世代グラフィックスAPI
 - glNextとして議論されていたもの
 - SDKツール、Window system integration(WSI)
- SPIR-V - グラフィックス、コンピュータの両方をサポートするシェーダ中間言語
 - Vulkan, OpenCL2.1の両方で使用





The Khronos 3D Ecosystem

Khronos: ソフトウェアとシリコンをつなぐ標準を策定する団体

ロイヤリティフリー、オープンスタンダードAPIを策定するオープンコンソーシアム

各プラットフォーム上のシリコンを動かすため下位APIのロードマップを定義

グラフィックス、コンピューティング、ビジョン処理分野で活動

ポータビリティを確保するための仕様定義及びコンFORMANCEテスト(認証テスト)を提供

業界による、業界のための、高性能API



Khronos のグラフィックス、コンピューティングAPI



次世代GPU向けAPIs



1つのOS



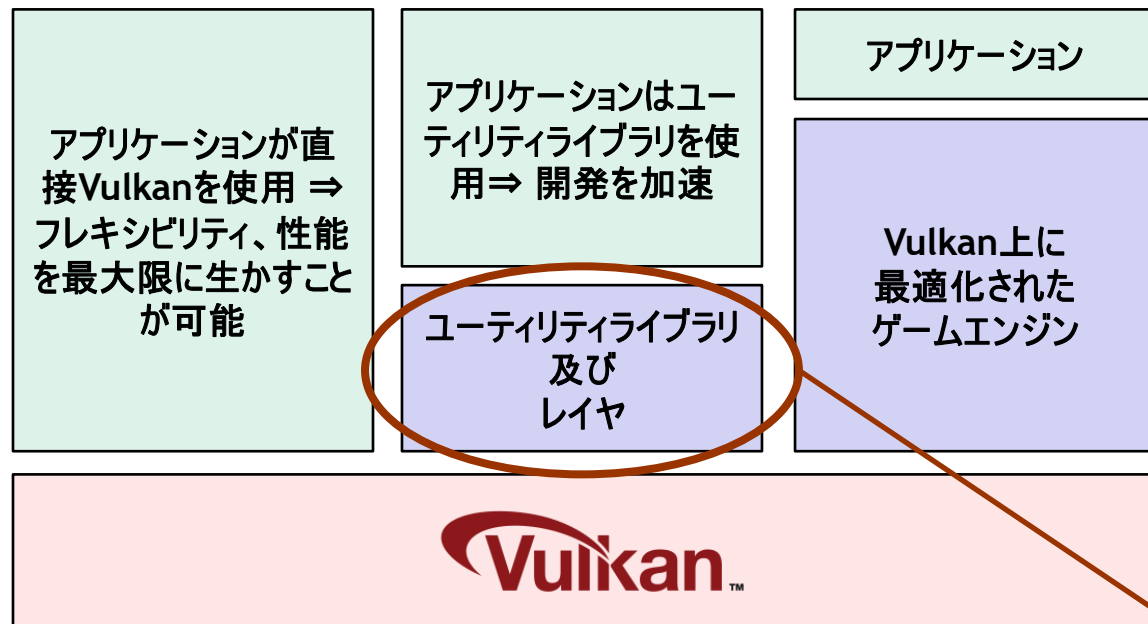
1つのプラット
フォーム・ベン
ダ



クロスプラットフォーム



エコシステムレイヤ



Vulkanワーキンググループにはゲーム、エンジンベンダが参加



Vulkanのエコシステムを使うにあたって、開発者は実装形態を選択可能

今後のイノベーション領域

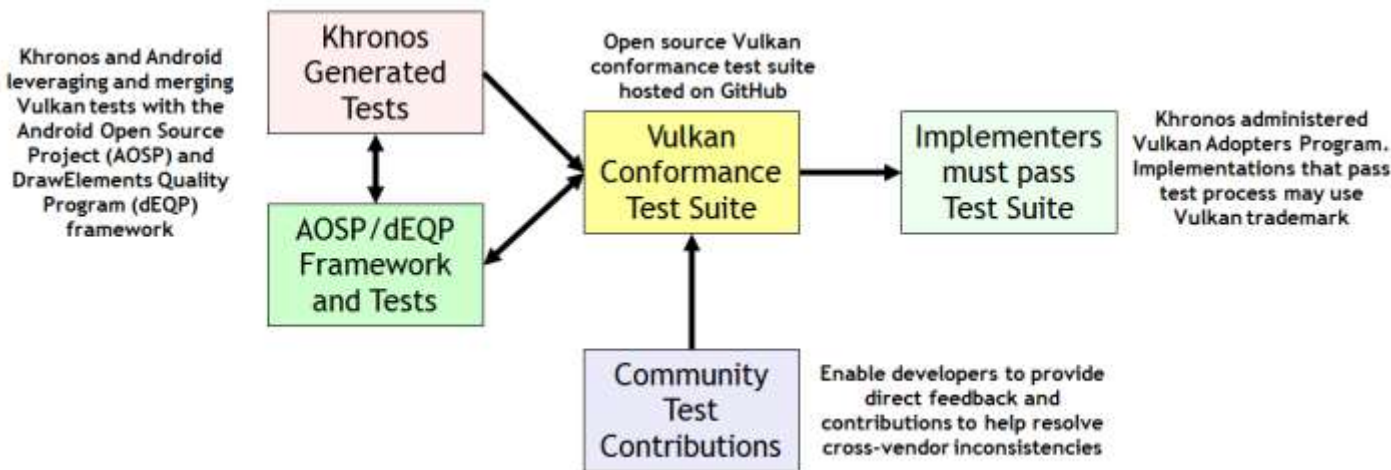
- 多くのユーティリティレイヤがオープンソースで登場
- OpenGLからの以降を容易にするレイヤ
- 特定ドメイン向けの環境提供

WebGLの様なエコシステムのダイナミクス

広く浸透、パワフル、ミドルウェアツール・ライブラリの多様性を可能にするフレキシブルなベースレイヤー

エコシステム構築と仕様策定を同時進行

- 開発者からのフィードバック及び、参加を可能にするVulkanテストパッケージのオープンソース化
- Khronosがローダー及び階層ツールアーキテクチャを提供
- オープンソースの階層ツール(layerd tool) - 初めにValve/LunarG
- フレキシブルなウィンドウシステムインテグレーション(WSI) - プラットフォームベンダと議論
- サンプルコード、ドキュメント、コースノート
- 革新的な中間言語 SPIR-V



SPIR-V による言語エコシステム

- 初めてのマルチAPI, 並列コンピューティング・グラフィックス向け中間表現(IR)
 - Vulkanシェーダ、OpenCLカーネルソース言語向けネイティブ表現
 - <https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf>
- クロスベンダ中間表現
 - 言語フロントエンドは複数ハードウェアを簡単にランタイムによりアクセス可能
 - 複数言語フロントエンドを用いたハードウェアアクセラレーション環境の活用
 - SPIR上でのプログラム解析、最適化ツール

開発者にとっての利点

複数プラットフォーム向けに同じフロントエンドが使える
カーネルのランタイムコンパイル時間を低減
シェーダ、カーネルのソースコードを提供する必要がなくなる
ドライバは、シンプルかつ信頼性向上が期待できる



SPIR-V オープンソースエコシステム

SPIR-V 仮仕様(V31)をSIGGRAPH2015でアップデート

Khronosはツール、トランスレータをオープンソース化する予定

GLSL

3rdパーティカーネル、シェーダ言語

OpenCL C

OpenCL C++

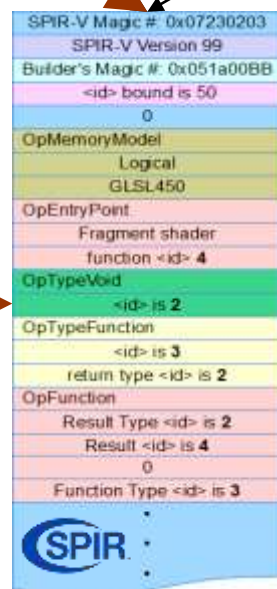
SPIR-V ツール

SPIR-V Validator

SPIR-V (Dis)Assembler

LLVM

LLVM から SPIR-V
への双方向の翻訳



その他中間表現

SPIR-V

- 32-bit Word Stream
- Extensible and easily parsed
- Retains data object and control flow information for effective code generation and translation

IHV Driver
Runtimes



10億台のデバイスから3Dグラフィックスにアクセス



300M Desktops / year
Windows, Mac, Linux



1.7B Mobiles / year



1B Browsers / year

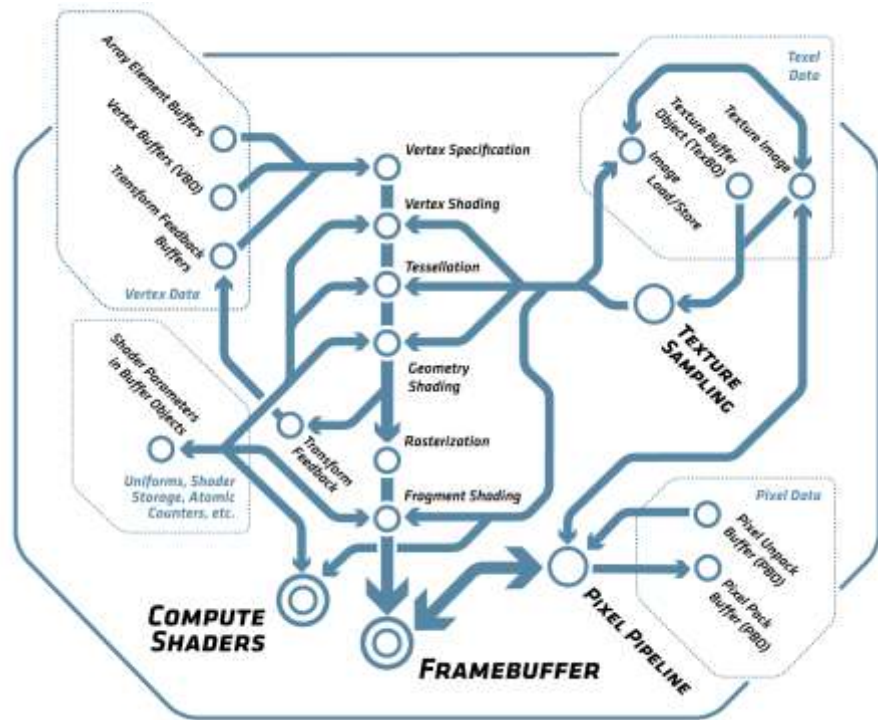


Worldwide Device Shipments by Segment (Thousands of Units)				
Device Type	2012	2013	2014	2015
PC (Desk-Based and Notebook)	341,273	299,342	277,939	268,491
Tablet (Ultramobile)	119,529	179,531	263,480	324,565
Mobile Phone	1,746,177	1,804,334	1,893,425	1,964,788
Other Ultramobiles (Hybrid and Clamshell)	9,344	17,195	39,636	63,835
Total	2,216,322	2,300,402	2,474,451	2,621,678

Source: Gartner (December 2013)

Source: Gartner (December 2013)

OpenGLにおける継続的な革新



Bringing state-of-the-art functionality to cross-platform graphics

Direct State Access (DSA)
Flush Control
Robustness
DX11 emulation
OpenGL ES 3.1 compatibility

OpenGL 4.5

OpenGL 4.4

OpenGL 4.3

OpenGL 4.2

OpenGL 4.1

OpenGL 3.3/4.0

OpenGL 3.2

OpenGL 3.1

OpenGL 2.0

OpenGL 2.1

OpenGL 3.0

2004

2005

2006

2007

2008

2009

2010

2011

2012

2013

2014

DirectX
9.0c

DirectX
10.0

DirectX
10.1

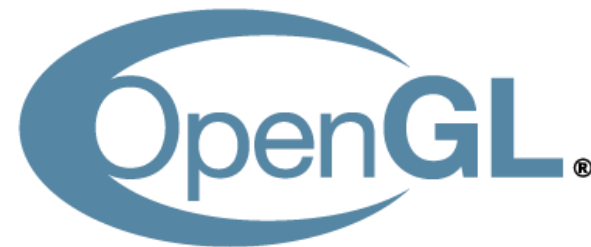
DirectX
11

DirectX
11.1

DirectX
11.2

SIGGRAPH 2015でアナウンスされた13の新OpenGL拡張

- **最新デスクトップグラフィックス機能のサポート**
 - 新機能が広く普及したタイミングで、OpenGLコアAPIに取り込みを行っていく
- **合理的なスパーステクスチャ(sparse texture)**
 - 複数のスパーステクスチャや大きなサイズのテクスチャの管理が可能
- **シェーダ拡張**
 - マルチパスアルゴリズムのためのインターロック、64bit整数型ハンドリング
 - 早期フラグメントテスト、拡張されたアトミックカウンタ、64bitタイミグカウンタ
- **マルチスレッドシェーダコンパイル**
 - アプリケーションのロード時間短縮
- **1ピクセル内におけるマルチサンプリング位置設定**
 - アンチエイリアスのクオリティ制御が可能に
- **OpenGL ES 3.2互換機能**
 - デスクトップOpenGL環境を使ったOpenGL ESアプリ開発



OpenGL ES ロードマップ

固定機能
パイプライン



プログラマブル
頂点・フラグメント
シェーダ



32bit整数型、浮動小数型
NPOT, 3D/depthテクスチャ
テクスチャアレイ
マルチレンダーターゲット



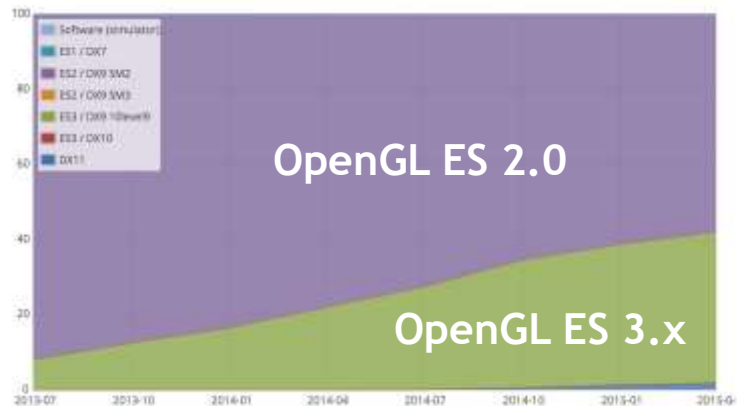
コンピュータシェーダ



テッセレーション・ジオメトリシェーダ、
ASTCテクスチャ圧縮
浮動小数点型レンダーターゲット デ
バグ・セキュア向けロバストネス



Epic's Rivalry demo using full Unreal Engine 4
<https://www.youtube.com/watch?v=jRr-G95GdaM>



<http://hwstats.unity3d.com/mobile/gpu.html>



2015年には17億以上のOpenGL ESサポートデバイス



Vulkan Overview

Vulkanプロジェクトステータス

- 2014年6月から8月にかけて
 - 次世代OpenGLプロジェクトの議論開始
 - 多くの業界関係者からコミットメント
 - 2014年のSIGGRAPHにてプロジェクトのアナウンス及び、参加者の募集をアナウンス
 - この時点ではプロジェクト名glNext
- 2015年3月 GDC 2015にて”Vulkan”を公開
- 2015年6月 ワーキンググループ内で仕様ソフトフリーズ
 - 仕様書準備
 - SDK構築
 - 認証テスト(Conformance test)準備
 - 仕様の詳細議論
- 2015年末の仕様書リリースに向けて議論進行中

Vulkanの方向性

- オープンスタンダード、クロスプラットフォーム、最新3D + コンピューティングAPI
 - OpenGLとの互換性は無し
 - ゼロベースで議論がスタート
- 目標
 - 整理された最新アーキテクチャ
 - マルチスレッド、マルチコアフレンドリー
 - CPUオーバーヘッドの大幅削減
 - アーキテクチャニュートラル: ダイレクトレンダラーだけでなくタイルベースのアーキテクチャもサポート
 - 明示的な制御によって性能を予測可能とする
 - 各実装間での信頼性、一貫性を改善

Vulkanのエッセンス

- 最新アーキテクチャ
 - これまでのGLコンテキスト⇒分割されたコマンドバッファとディスパッチキューに置き換え
 - コマンドレベルでのハンドリング環境を提供
- スレッドフレンドリ
 - ほとんどのオブジェクトタイプはスレッドフリー
 - 同期制御はアプリケーションの責任
- 低CPUオーバーヘッド
 - エラーチェック、依存関係のトラッキングはアプリケーションの仕事
 - Validation layer(検証レイヤ)にて最適化可能
- いつ処理が終わるか明示的にコントロール(explicit control)
 - シェーダコンパイル、コマンド生成がいつ行われるかが明確にする
 - 変わらないステートは早期に定義(決着)する事で、ドライバからディスパッチ時間を削除

ちなみに

- VulkanはAMDのマントルをベースに仕様策定しています。





Vulkan API

Vulkan API一式の説明

- Instance: インスタンスの定義からスタート(アプリケーションの情報、Callbackの定義)
- GPU device: GPUハードウェアの宣言、クエリ
- Queue: コマンドを入れるためのキュー
- Command buffer: コマンド自体
- Shader: プリコンパイルされたシェーダ
- Pipeline state: パイプライン上のステート定義
- Resource/GPU memory: リソースのアロケーション => リソースはdescriptorで定義
- Render pass: レンダリングパスの定義
- Drawing: レンダリングパス内の描画実行
- Synchronization: イベントプリミティブを使った同期
- Resource state: コマンドバッファ上命令はパイプラインバリアで区分される
- Work: コマンドバッファキューで実行されるもの
- Presentation: 表示
- Object destruction: オブジェクトの破棄管理

詳細はBackupスライド参照

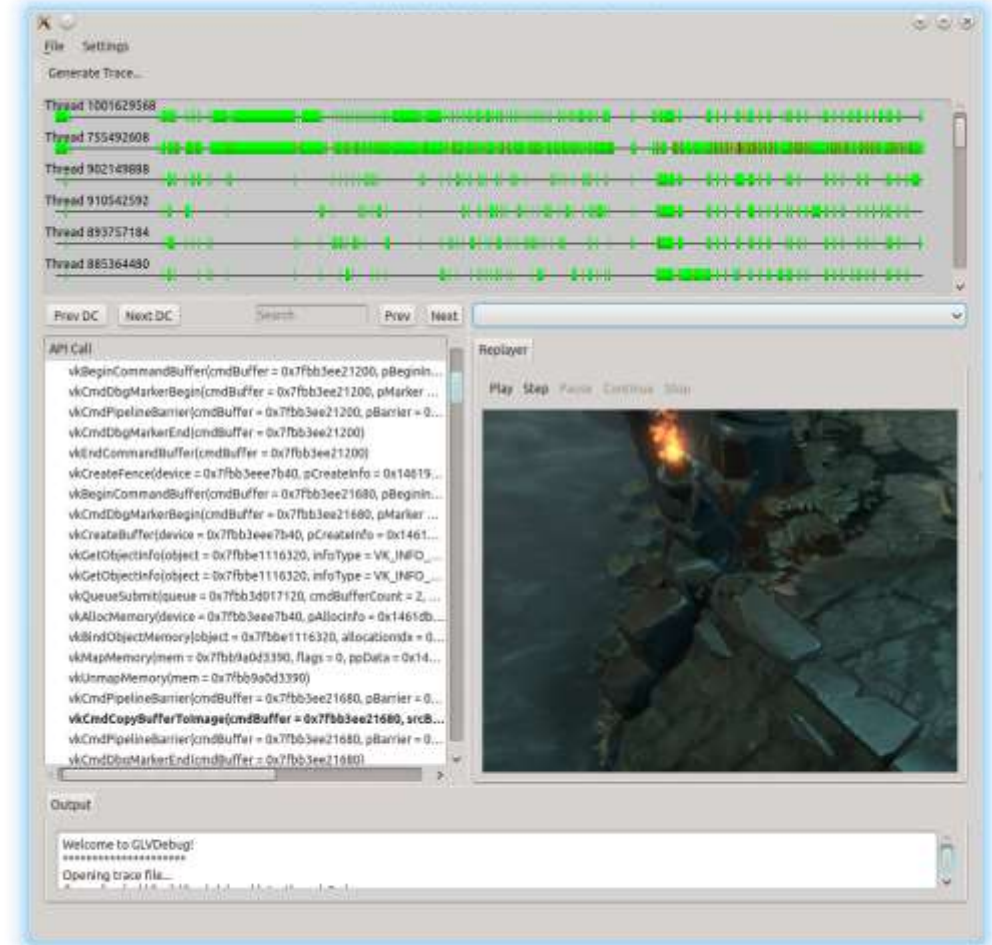


Vulkan SDK

LunarG SDK

Prototype Vulkan Debugger from Valve and LunarG
LunarG.com/Vulkan

- ValveがLunarGにファンディングし、Vulkanのエコシステムをサポート
 - オープンソースベースクロスプラットフォームツール
 - LunarGがSDKを提供し、LunarXchangeにサポート
 - <http://lunarg.com/vulkan>





<http://LunarG.com/Vulkan/>

Vulkan Loader

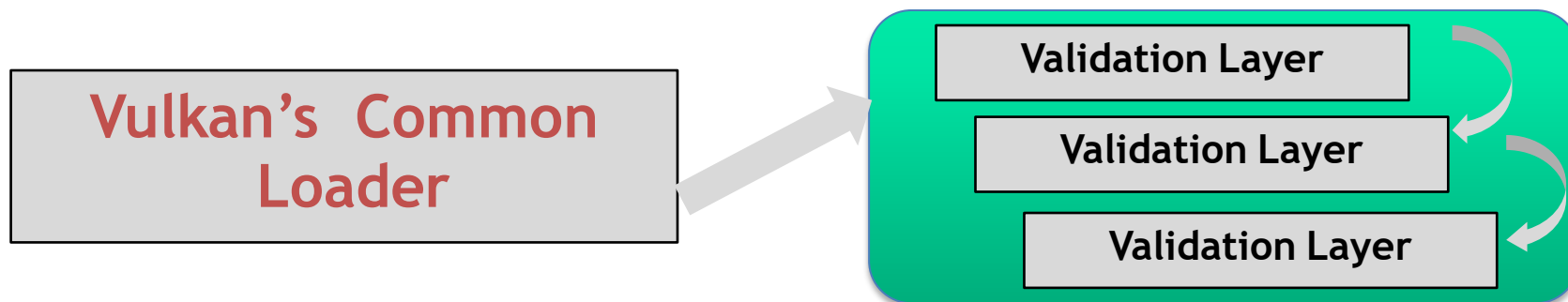
- レイヤー群(検証、デバグレイヤ)を使うための共通ローダー
 - アプリやICD(Installable Client Driver) に影響しない形でレイヤーを追加
 - プラグアンドプレイベースの環境
 - 例えば、複数のVulkanドライバがシステム上に共存可能
 - 複数ベンダからの複数ドライバをまとめられる
- 以下のハンドラ
 - ドライバ管理
 - レイヤライブラリ
 - インスタンス拡張

e.g. LunarG Vulkan SDKレイヤー群

Layer名	内容
APIDump	APIコール、パラメータ、値のプリントAPI
DrawState	ディスクリプタ群、パイプラインステート、ダイナミックステートの検証レイヤ
Image	テクスチャフォーマット、レンダーターゲットフォーマットの検証レイヤ
MemTracker	GPUメモリ、オブジェクト・コマンドバッファのバインド状況のトラック、検証
ObjectTracker	すべてのVulkanオブジェクト、フラグ無効オブジェクト、オブジェクトメモリリークのトラッキング
ParamChecker	APIパラメータ値の検証レイヤ
ShaderTracker	SPIR-Vモジュールとグラフィックスパイプラインのインターフェイス検証
Threading	マルチスレッドAPIの使い方の検証

Vulkan Loader

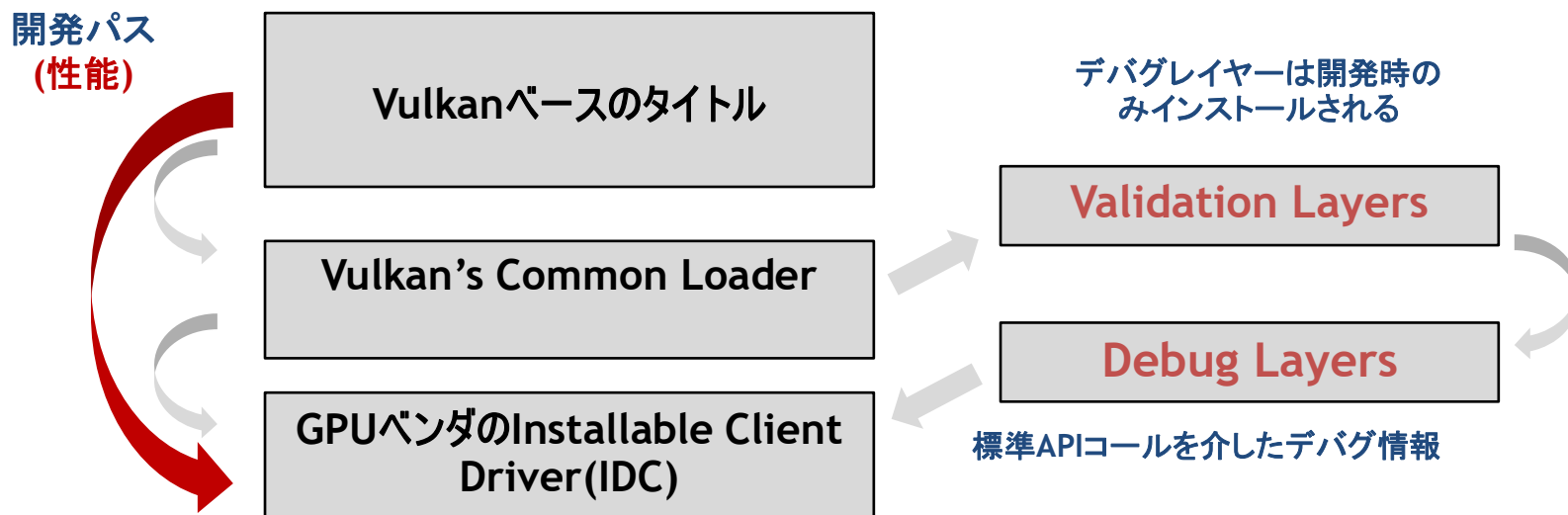
- ローダはレイヤーのみサポート。拡張は生成時に有効化
 - レイヤーは明示的にも暗示的にも有効化が可能
- ローダはレイヤーの呼び出しシーケンスを制御
 - 所望のシーケンスでレイヤーのチェーンを作成
 - レイヤー間で次どこにジャンプするかを定義
 - 複数のドライバクエリをまとめる
- GPUハードウェアから見たローダー
 - アプリケーションの挙動をキャプチャ、デバッグするのに有効
 - ドライバで無駄なエラーチェックを減らせる



レイヤー向けVulkanツールアーキテクチャ

- フレキシブルなレイヤーベースのデザイン

- 開発時、共通かつ拡張可能な環境下で、最終製品への性能に影響なしにコード検証、デバッグプロファイリングが可能





Vulkan向け中間表現 SPIR-V



Standard
Portable
Intermediate
Representation

目指すところ:

- 1) GPU, 並列コンピューティング向けシェーダ、コンピュートカーネルのポータブルなバイナリ表現
- 2) OpenCL C/C++, GLSL, その他シェーディング言語がターゲット

ポータブルなシェーダ表現による、
コンパイラのエコシステム構築

なぜSPIRを使うか

SPIRがなかった場合:

- ベンダーはソースで提供する必要
 - 知財・ノウハウ流出のリスク
- 制約されたポータビリティ
 - SWベンダからはフロントエンド以上を超えた使用ができない
 - ベンダー毎に別々のフロントエンドセマンティクス
- ランタイム時にコンパイルが必要で、時間がかかる

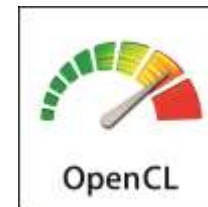
SPIRがあった場合:

- 1つのバイナリファイルを提供
 - 解読にツールが必要; 知財・ノウハウの保護
- ポータビリティの改善
 - SWベンダは自分のフロントエンド、ツールチェーンを構築可能
 - 複数のSWベンダで、共通のフロントエンドを共有可能
- ランタイム時、コンパイル時間の節約
 - いくつかの段階がオフロードされる

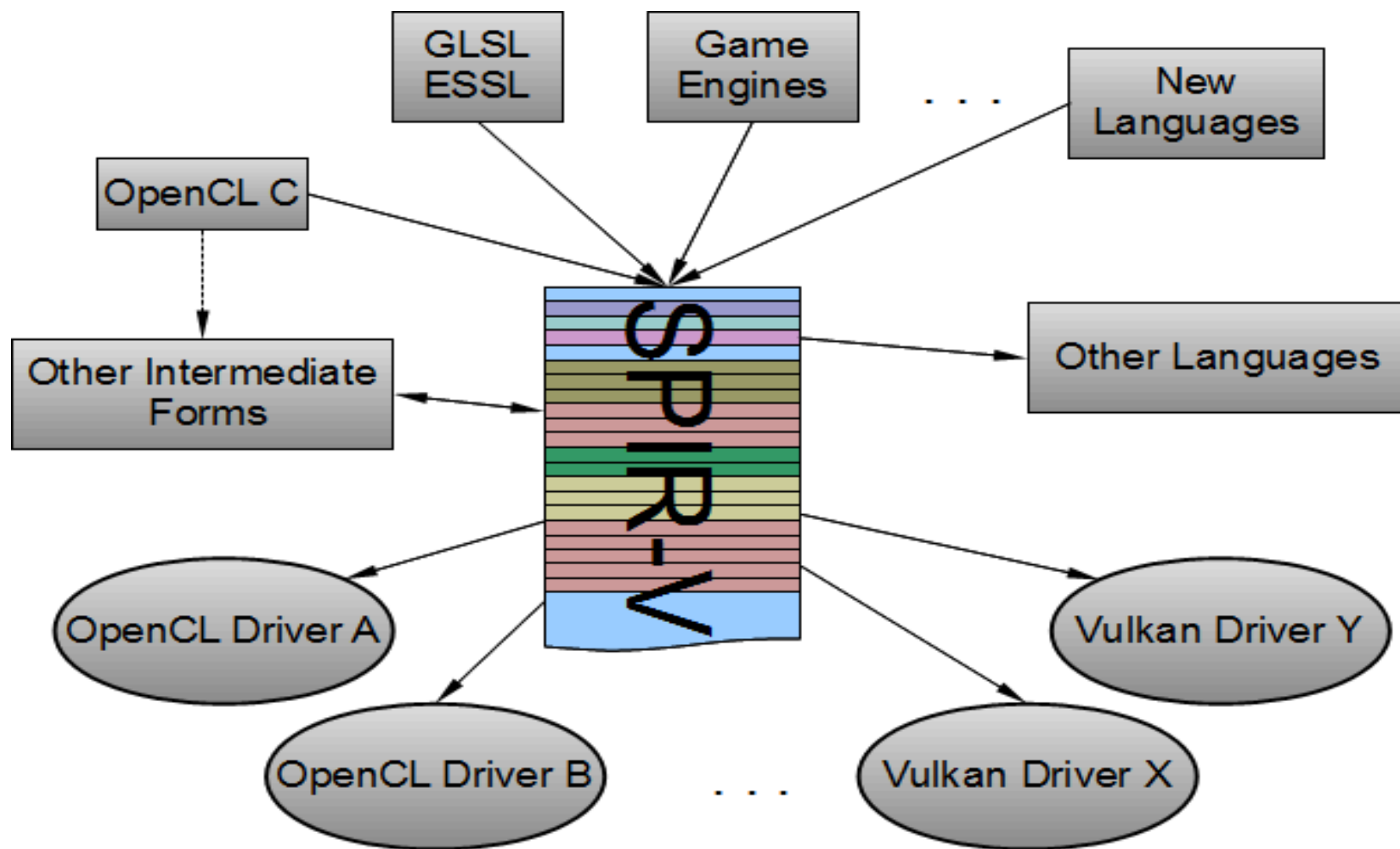
特定分野向け言語、C++コンパイラなど
イノベーションの機会・余地が提供される

SPIR-Vとは

- Khronosグラフィックス、コンピュータAPIへの入力向け新しい中間言語
 - Khronos仕様として策定
 - Khronosグラフィックス、コンピュータのイデオムをそのまま表現可能
 - 例: コントロールフロー制約を持った暗示的派生
 - GLSL, OpenCL高級言語向けメモリ、実行モデル
- Vulkan向けコア
 - APIがサポートする唯一の言語
 - Vulkan向け明示的なマシンモデル
 - GLSL/ESSLシェーダ言語を完全サポート
 - SPIR-V向けにその他言語も使用可能
- OpenCL 2.1向けコア
 - OpenCL 1.2, 2.0, 2.1カーネル言語をサポート



共通中間表現としてのSPIR-V



Vulkanサマリ

- Vulkanは単なるローレベルではない - 最新ハードウェアのよりよい抽象化
- Vulkanは超低オーバーヘッド
 - CPUの負荷を低減する ⇒ アプリケーションがよりCPUの演算パワーを使用可能
 - 明示的なスレッドサポート ⇒ グラフィックスAPIのことを気にせず多くのスレッドを使用可能
 - 1度コマンドバッファの作成をすれば、何回でもリユース可能
- クロスプラットフォーム、クロスベンダー
 - 1つのOSやOSバージョンに縛られない
 - 1つのGPUファミリーやベンダーに縛られない
 - 1つのアーキテクチャに縛られない
 - デスクトップ + モバイル、フォワードやディファード、タイルやノンタイル
- オープンかつ拡張可能
 - Khronosの枠組みによるオープン化
 - 様々なインダストリ、ゲーム、CAD, AAA+casualなど幅広い分野でのコラボレーション
 - 拡張機能、レイヤ、デバガ、ツールのフルサポート
 - 完全にドキュメント化されたSPIR-V(中間言語) コンパイラの自作も可能



Thank you

Vulkan詳細

Vulkanコーディング: 事始め

- Vulkanはインスタンス(instance)として記述
- アプリケーションは複数のVulkanインスタンスを持てる
 - それぞれは独立
 - ミドルウェア化、サブシステム化等を容易にする
- インスタンスはローダで所有
 - マルチベンダの複数ドライバをハンドリング
 - 複数GPUのハンドリングを行う
 - 複数ドライバを作成 ⇒ 複数GPUをサポートする1つの大きなドライバに見せる

本スライドに書かれているCodeは最終的な仕様の関数名、用例ではなくあくまでPseudo codeになります。

```
VK_APPLICATION_INFO appInfo = { ... };  
VK_ALLOC_CALLBACKS allocCb = { ... };  
VK_INSTANCE instance;
```

```
vkCreateInstance(&appInfo, &allocCb, &instance);
```


Vulkanにおける複数GPUサポート

- Vulkanインスタンス作成の際の定義情報
 - アプリケーション情報 - 作成するユーザVulkanアプリケーションに関する情報
 - コールバックのアロケーション - Vulkanはユーザアロケータでシステムメモリをアロケーション
- インスタンス作成をしたら、GPUsについて確認

```
uint32_t gpuCount;  
VK_PHYSICAL_GPU gpus[10];
```

```
vkEnumerateGpus(instance, ARRAYSIZE(gpus), &gpuCount, gpus);
```

- GPUのリスト作成及び、カウント
- 別々のベンダーのGPUでもOK
 - CPUに統合されたGPU + GPUカードの構成
 - 複数GPUカード構成のシステム
 - API上でのGPU間でのリソースシェア及び、明示的な複数GPUサポート

Vulkan GPUクエリ

- GPUの情報クエリ

```
VK_SOME_GPU_INFO_STRUCTURE info;  
uint32_t infoSize = sizeof(info);
```

```
vkGetGpuInfo(gpu[0], VK_GPU_INFO_WHATEVER, &infoSize, &info);
```

- GPUに関していろんな情報が取得可能

- メーカー、相対的な性能、メモリサイズ、キュータイプなど

- 複数GPU間での互換性クエリ

```
VK_GPU_COMPATIBILITY_INFO compatInfo;
```

```
vkGetMultiGpuCompatibility(gpuA, gpuB, &compatInfo);
```

- GPU間に関する互換情報に含まれるもの

- 完全リソースシェア可能 or いくつかのリソースについてシェア可能 or 互換性はなし
といったオプション情報を取得

Vulkanデバイス

- 1つのGPUから1つのデバイスインスタンスを生成

```
VK_DEVICE_CREATE_INFO info = { ... };  
VK_DEVICE device;
```

```
vkCreateDevice(gpu, &info, &device);
```

- VK_DEVICE_CREATE_INFOは以下の情報が含まれる
 - 必要なキューの数、型
 - 使いたい拡張仕様
 - 検証(Validation)のレベル
 - ドライバでのエラーチェックは基本無し
 - 各レイヤー上でそれぞれのレベルでの検証を行う
 - ドライバが各ベンダー特有の挙動を各レイヤーで検証する

Vulkanキュー

- デバイスからキューハンドラを取得

```
VK_QUEUE queue;
```

```
vkGetDeviceQueue(device, 0, 0, &queue);
```

- キューは2つのインデックスで表現
 - ノード順(Node ordinal)
 - 互換性を持つキュー群。キューノードがずらずら並ぶイメージ
 - キューインデックス(Queue index)
 - 複数のキューインスタンス。互換性を持たないばらばらのキューをインデックスで指定
- キューのカプセル化
 - 機能毎 - グラフィックス、コンピュート、DMA
 - スケジューリング毎 - 独立のスケジューリング、非同期

Vulkanコマンドバッファ

- GPUコマンドをコマンドバッファに投入

```
VK_CMD_BUFFER_CREATE_INFO info;  
VK_CMD_BUFFER cmdBuffer;
```

```
vkCreateCommandBuffer(device, &info, &cmdBuffer);
```

- 必要な数のコマンドバッファを作成可能
- コマンドバッファ作成時の情報
 - どのキュー群にコマンドを投入したいか (ノード順(node ordinal))
 - どのくらいアグレッシブにドライバが最適化していいかの情報(最適化レベル⇒詳細議論中)など

Vulkanコマンド

- コマンドバッファへのコマンド追加

```
VK_CMD_BUFFER_BEGIN_INFO info = { ... };  
vkBeginCommandBuffer(cmdBuf, &info);
```

```
vkCmdDoThisThing(cmdBuf, ...);  
vkCmdDoSomeOtherThing(cmdBuf, ...);
```

```
vkEndCommandBuffer(cmdBuf);
```

- ユーザーが行うべきもの

- コマンドバッファの再利用
- この段階でコマンド投入の最適化を行う。(ドロー直前には行わない)
- 変更されていないステートをうまくとりまとめる、ステート変更に係わる処理時間を減らす

Vulkanシェーダ

- Vulkanシェーダの事前コンパイル(ランタイムソースコードコンパイルは行わない)

```
VK_SHADER_CREATE_INFO info = { ... };  
VK_SHADER shader;  
  
vkCreateShader(device, &info, &shader);
```

- シェーダ生成情報に含まれるもの
 - シェーダソースのポインタ
 - SPIR-V - ポータブルで、ベンダー共通で、オープンで、拡張可能なシェーダバイナリ
 - その他のIRも同じインターフェイスでサポート可能
 - その他追加オプション情報
- 複数スレッドからシェーダをコンパイル
 - 正しくコンパイルが行われるよう、ドライバがいろいろ頑張る

Vulkanパイプラインステート

- パイプラインステートもコンパイル済みのものを入力

```
VK_GRAPHICS_PIPELINE_CREATE_INFO info = { ... };  
VK_PIPELINE pipeline;
```

```
vkCreateGraphicsPipeline(device, &info, &pipeline);
```

- 生成情報に含まれるもの
 - コンパイル済みシェーダ
 - ブレンド、デプス、カリング、ステンシルステートなど
 - ステートリスト(可変mutable)である必要)

- パイプラインはシリアライズ、デシリアライズ化のいずれも可能

```
uint32_t dataSize = DATA_SIZE;  
void* data = malloc(DATA_SIZE);
```

```
vkStorePipeline(pipeline, &dataSize, data);
```

```
...
```

```
vkLoadPipeline(device, dataSize, data, &pipeline)
```

Vulkan可変(mutable)ステート

- いくつかのパイプラインステートはミュータブル or ダイナミック(動的)
- より小さなステートオブジェクトで表現

```
VK_DYNAMIC_VP_STATE_CREATE_INFO vpInfo = { ... };  
VK_DYNAMIC_VP_STATE_OBJECT vpState;
```

```
vkCreateDynamicViewportState(device, &vpInfo, &vpState);
```

```
VK_DYNAMIC_DS_STATE_CREATE_INFO dsInfo = { ... };  
VK_DYNAMIC_DS_STATE dsState;
```

```
vkCreateDynamicDepthStencilState(device, &dsInfo, &dsState);
```

Vulkanリソース

- メモリリソースは1個のCPUと1個のGPUコンポーネントを持つ
- CPU側は、vkCreate*関数でアロケート

```
VK_IMAGE_CREATE_INFO imageInfo = { ... };  
VK_IMAGE image;  
vkCreateImage(device, &imageInfo, &image);
```

```
VK_BUFFER_CREATE_INFO bufferInfo = { ... };  
VK_BUFFER buffer;  
vkCreateBuffer(device, &bufferInfo, &buffer);
```

- GPUのメモリアロケーションはアプリケーションの責任

Vulkan GPUメモリ

- 確保するメモリのためのクエリオブジェクト

```
VK_IMAGE_MEMORY_REQUIREMENTS reqs;  
size_t reqsSize = sizeof(reqs);  
  
vkGetObjectInfo(image,  
                 VK_INFO_TYPE_IMAGE_MEMORY_REQUIREMENTS,  
                 &reqsSize, &reqs);
```

- アプリケーションがGPUメモリアロケーションを行う

```
VK_MEMORY_ALLOC_INFO memInfo = { ... };  
VK_GPU_MEMORY mem;  
vkAllocMemory(device, &memInfo, &mem);
```

- アプリケーションが確保した上記GPUメモリをオブジェクトにバインド

```
vkBindObjectMemory(image, 0, mem, 0);
```

Vulkanディスクリプタ

- Vulkanシェーダリソース(ユニフォームバッファ、サンプラなど)はディスクリプタで表現
- ディスクリプタはセット群(sets)によりアレンジされる
- セット群(Sets)はプール群(pools)からアロケートされる ⇒ この辺りの構成はWGでも議論中
- それぞれのセットはレイアウト(layout)(パイプライン作成時に分かるもの)を持つ
 - レイアウトはセット群とパイプライン群でシェアし、それぞれが合致する必要
 - レイアウトはオブジェクトで表現し、パイプライン作成時に渡す
- パイプライン群(同じレイアウトのセット群を使用)はスイッチ可能
- 様々なレイアウト群の膨大なセット群を1つのパイプライン上の1個のチェーン(chain)でサポート

```
vkCreateDescriptorPool (...);  
vkCreateDescriptorSetLayoutChain (...);  
vkCreateDescriptorSetLayout (...);  
vkAllocDescriptorSets (...);
```

Vulkanレンダーパス

- レンダーパスは1フレーム内の段階(logical phases)で表現
- レンダーパスは明示的なオブジェクト

```
VK_RENDER_PASS_CREATE_INFO info = { ... };  
VK_RENDER_PASS renderPass;
```

```
vkCreateRenderPass(device, &info, &renderPass);
```

- レンダーパスにはレンダリングについて多くの情報が含まれる
 - レイアウト(layout)とフレームバッファ種別のアタッチメント
 - レンダーパスをどのタイミングでスタート・エンドさせるか
 - レンダーパスでエフェクトをかけるフレームバッファの領域
- タイルベース、ディファードレンダラにとってきわめて重要な情報
 - フォワードレンダラ向けにも有効な情報

Vulkan描画(Drawing)

- 描画はレンダーパス内で実行
- コマンドバッファのコンテキストとしてDraw実行が格納される

```
VK_RENDER_PASS_BEGIN beginInfo = { renderPass, ... };
```

```
vkCmdBeginRenderPass(cmdBuffer, &beginInfo);
```

```
vkCmdBindPipeline(cmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline);  
vkCmdBindDescriptorSets(cmdBuffer, ...);  
vkCmdDraw(cmdBuffer, 0, 100, 1, 0);
```

```
vkCmdEndRenderPass(cmdBuffer, renderPass);
```

- パイプライン、ダイナミックステートオブジェクト、その他リソースがすべてコマンドバッファ内に含まれる
- すべての描画タイプをサポート
 - インデックス、非インデックス、ダイレクト、(複数)インダイレクト、コンピュータディスパッチ等

Vulkan同期化(Synchronization)

- イベントプリミティブを使って同期化

```
VK_EVENT_CREATE_INFO info = { ... };  
VK_EVENT event;
```

```
vkCreateEvent(device, &info, &event);
```

- イベントとしてセット、リセット、クエリ、ウェイトが使用可能

```
vkSetEvent(...);  
vkResetEvent(...);  
vkGetEventStatus(...);  
vkCmdSetEvent(...);  
vkCmdResetEvent(...);  
vkCmdWaitEvents(...);
```

- コマンドバッファは実行完了をイベントとして通知できる

Vulkanリソースステート

- コマンドバッファ上の操作をパイプラインバリアによって区分
- バリアはウェイト、イベント通知が可能
- バリアはステートからステートにリソース遷移できる
 - 描画可能
 - テクスチャとしての読み込み
など

```
VK_IMAGE_MEMORY_BARRIER imageBarrier = { ... };  
VK_PIPELINE_BARRIER barrier = { ..., 1, &imageBarrier };  
  
vkCmdPipelineBarrier(cmdBuffer, &barrier);
```

- ドライバはステートのトラッキングに関与しない
 - ステートのトラッキングはアプリケーションの責任
 - もし、アプリが間違うと、不正描画やクラッシュを引き起こす
 - 検証レイヤ(Validation layer)でステートのトラック(性能低)して、デバグを行う

VulkanワークのQueue投入

- ワーク(Work)はデバイスに属する各キュー(Queue)上で実行される
- 完成されたコマンドバッファは実行のためにキューに投入される

```
VK_CMD_BUFFER commandBuffers[] = { cmdBuffer, ... };  
vkQueueSubmit(queue, 1, commandBuffers, fence);
```

- キュー自体は、自身のメモリを持つ
 - ドライバはアプリのキューメモリのトラッキングは行わない

```
vkQueueAddMemReference(queue, mem);  
vkQueueRemoveMemReference(queue, mem);
```

- キューはオブジェクトのオーナーシップのためにセマフォによる通知、ウェイトを行える

```
vkQueueSignalSemaphore(queue, semaphore);  
vkQueueWaitSemaphore(queue, semaphore);
```

Vulkan プレゼンテーション

- プレゼンテーション(Presentation)でどうやって画面表示をさせるかを定義
- 表示可能デバイスは特別な種類のイメージとして表現
 - フレームバッファバッファにバインド可能
 - WSI (Window System Interface)と呼ばれるプラットフォーム依存ジュールにより作成
- 少数(-2?)のWSIバインドを定義
 - コンポジションシステム(合成処理部分が(Compositor)が自分の表示可能サーフェイスを持つ)
 - アプリサーフェイス表示: アプリケーションが持っているサーフェイスのプレゼンテーション(表示)を許すシステム
- WSIは以下のハンドリングも行う
 - 表示デバイス数、ビデオモード数のハンドリング
 - フルスクリーン化
 - VSyncの制御
- プレゼンテーションもコマンドバッファ内で定義

Vulkan オブジェクト破棄

- アプリケーションはオブジェクトの破棄の責任をもつ
 - 正しくオーダハンドリングを行う必要
 - 参照カウンタはなし
 - 暗示的なオブジェクトのライフタイムはなし
- 使用中のオブジェクトの削除禁止
- 多くのオブジェクトのデストロイは以下の関数を使用

```
vkDestroyObject(object);
```

- いくつかのオブジェクトは専用関数を使用

```
vkDestroyDevice(device);  
vkDestroyInstance(instance);
```

Vulkan AZDO(Approaching Zero Driver Overhead)?

- VulkanはすでにPDCTZO (Pretty Darn Close to Zero Overhead: ものすごくオーバーヘッドZERO)
 - アプリがオプションを検証レイヤを入れない限り、ほとんど検証オーバーヘッドはなし
 - アプリがすべて管理 - 基本ドライバは何もやらない
 - ハードウェアのよりよい抽象化 - APIからハードウェアのマッピングの容易化
- 作成済みコマンドバッファを何度も使用可能
 - コマンドバッファ作成コストは0になる
- バインド不要
 - 必要な議論の余地 - ディスクリプタセットは任意サイズ
 - 明示的なメモリ配置が前提
- Sparse
 - サポート
- MultiDrawIndirect
 - サポート
- Shader Draw Parameters
 - サポート

GDC以降の変更サマリ

- 開発者向け改善

- プログラマ向けコンパイル時の改善 (型、const)
- メモリのヒープサポート

- 機能の拡張

- パイプラインキャッシュ
 - 効率的なパイプライン作成、スイッチ機構
 - パイプラインでのコンポーネントのシェア
 - セーブ・リストア機能
- コマンドバッファ管理
 - コマンドプール: コマンド作成の効率化 / 複数モード準備
 - 2レベルコマンドバッファ: グループ単位でのコマンド呼び出し
- マルチレンダーパース
 - 上記2レベルコマンドバッファのサポート



SPIR-V: もう少し詳細

- バイナリ中間言語
 - 32bitワードのリニアなストリーム
- モジュール内の機能は、基本ブロックのCFG(control-flow graph)で格納
- ロード・ストア命令は宣言された変数へのアクセスとして使われる
- 中間結果はSSA(Single Static-Assignment)で表現される
- データオブジェクトは階層的なタイプを持ち論理表現される
 - e.g. 集合の未フラット化 or 物理レジスタへのアサイン
- 選択可能なアドレッシングモデル
 - ポインタ的使い方や固定メモリモデルをサポート
- 容易に拡張可能
- SPIR-Vモジュールのセマンティックを変更することなしで安全に中身を確認できる様なデバッグ情報のサポート

構造的な制御フロー

```
for (...) {
```

```
    if (...)
```

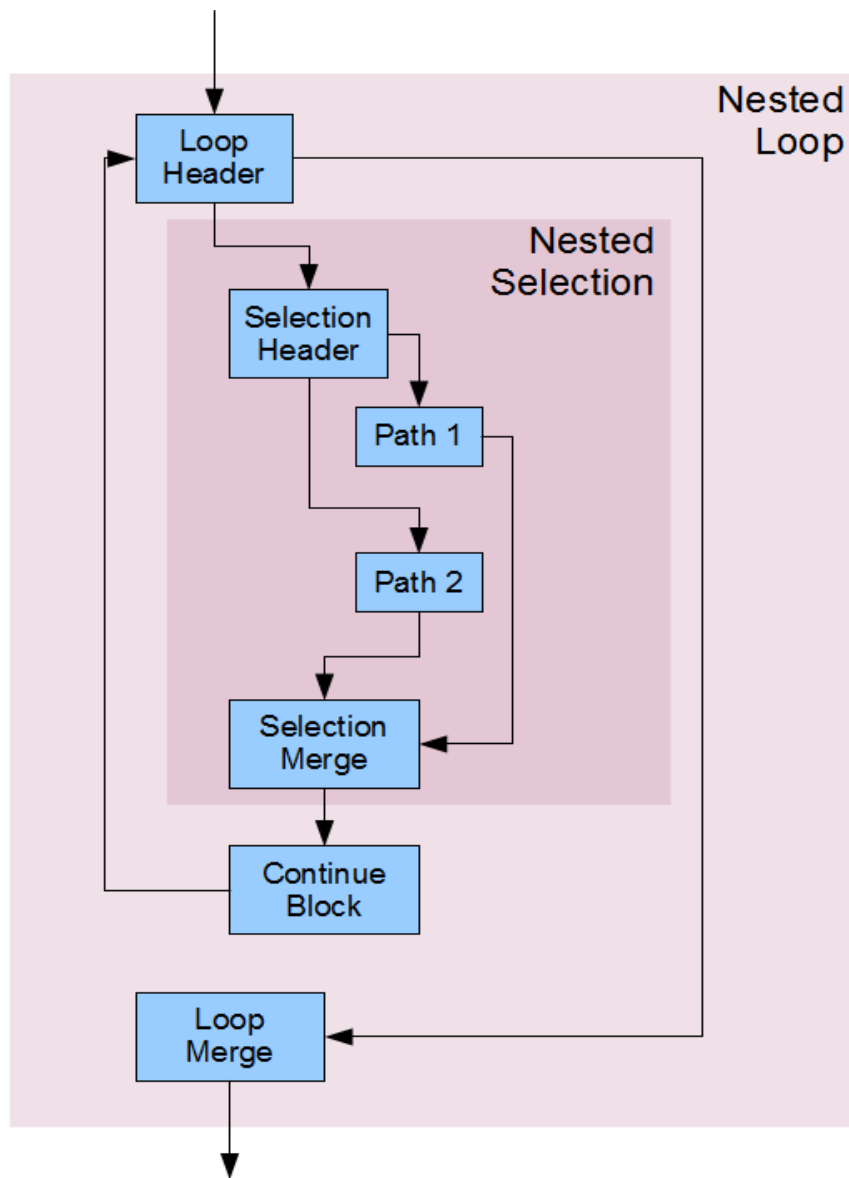
```
        ...
```

```
    else
```

```
        ...
```

```
    ...
```

```
}
```



```
11: Label
```

```
...
```

```
LoopMerge 12 NoControl
```

```
BranchConditional 18 19 12
```

```
19: Label
```

```
22: ...
```

```
SelectionMerge 24 NoControl
```

```
BranchConditional 22 23 28
```

```
23: Label
```

```
...
```

```
Branch 24
```

```
28: Label
```

```
...
```

```
Branch 24
```

```
24: Label
```

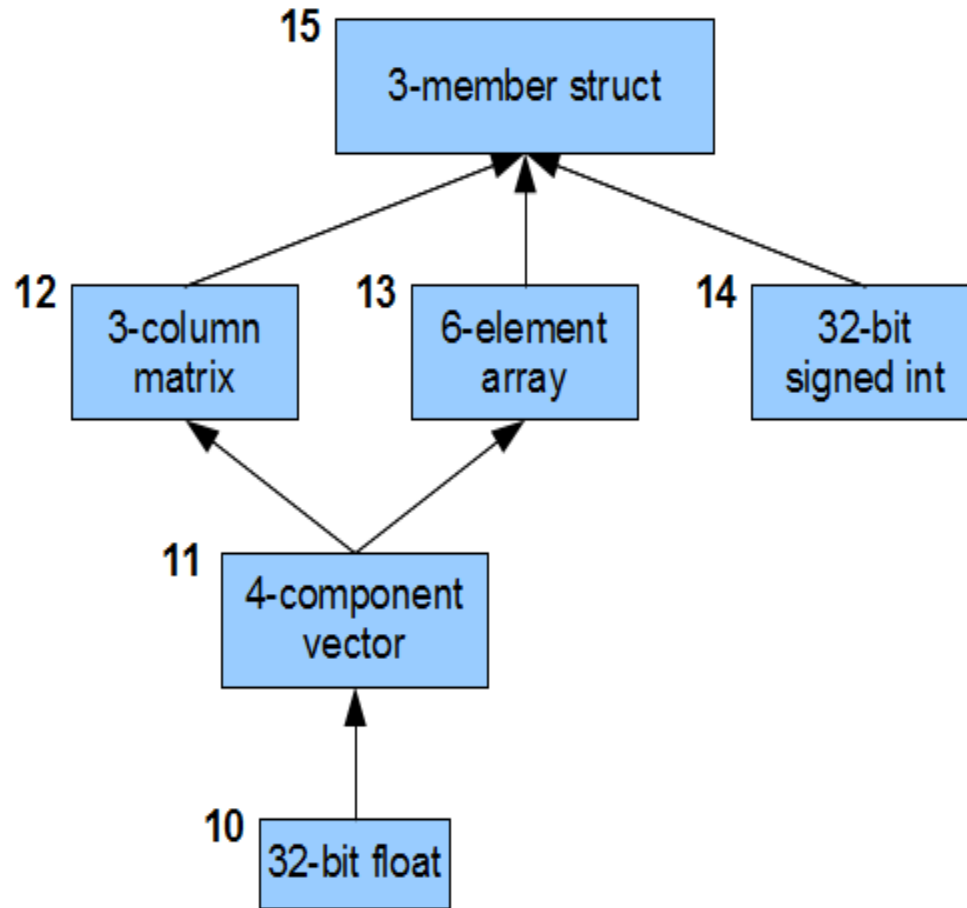
```
...
```

```
Branch 11
```

```
12: Label
```


階層的な型、コンスタント、オブジェクト

```
struct {  
    mat3x4;  
    vec4[6];  
    int;  
};
```



10: OpTypeFloat 32
11: OpTypeVector 10 4
12: OpTypeMatrix 11 3
13: OpTypeArray 11 6
14: OpTypeInt 32 1
15: OpTypeStruct 12 13 14