KENWRIGHT

INTRODUCTION TO COMPUTER GRAPHICS AND THE VULKAN API

TECHNICAL BOOK

Introduction to Computer Graphics and the Vulkan API *Kenwright*

.

BOOK TITLE:
Introduction to Computer Graphics and the Vulkan API
ISBN-13: 978-1-548-61617-5
ISBN-10: 1-548-61617-6

Revision: 035628082017

First published, July 2017

Contents

Dedicated to those who appreciate the beauty and complexity of computer graphics

1

Introduction & Overview

1.1 Getting Started

This book provides an introductory guide to getting started with computer graphics using the Vulkan API. The book focuses on the practical aspects with details regarding previous and current generation approaches, such as, the shift towards more efficient multi-threaded solutions. The book has been formatted and designed, so whether or not you are currently an expert in computer graphics, actively working with an existing API (OpenGL), or completely in the dark about this mysterious topic, this book has something for you. If you're an experienced developer, you'll find this book a light refresher to the subject, and if you're deciding whether or not to delve into graphics and the Vulkan API, this book may help you make that significant decision. This is an ambitious book, but not unrealistic, and we know that computer graphics is a little bit of an art and involves a variety of skills and abilities. There is so much more to know than this book is able to present - however, it presents the essential facts of the subject with a high-level introduction to the core components and their mechanics. It's not that we necessarily excluded anything critical from this book, but it would be unrealistic to try and cover every possible aspect in a single text. For the sake of practicality, we discuss a variety of important aspects of the Vulkan API, such as, the differences between traditional graphical API paradigms, setting up a Vulkan project, performance factors and real-world applications and examples.

Sample program listings and support material are provided online to complement the book.

Figure 1.1: Designed and maintained by Khronos Group for high performance on rendering and compute [6].

1.2 Computer Graphics

Computer graphics is an exciting and important multi-discipline subject with applications in:

- visualisation solutions,
- video games,
- image and video processing,
- graphical modeling,
- animation,
- augmented and virtual reality,
- production/tool optimisation (CPU/GPU),
- real-time solutions,
- rendering & simulation,
- visual effects,
- user interaction
- robotics
- ...

Computer graphics covers topics from extraction and visualisation to generation and manipulation in both 2-dimensional and 3-dimensional contexts. In this book, you'll focus primarily on 3-dimensional visual solutions. However, you'll still require and apply 2-dimensional principles like texture manipulation and mapping to pixel and screen space effects (e.g., blurring, edge detection and smoothing). You'll discover that computer graphics gives you the power to create worlds of infinite possibilities (e.g., from chocolate cities 'choco-land' to real-world locations like London) or help visualise complex problems (like structural stress in buildings or the workings of internal organs in the human body). The implementations can range in complexity as well - from a simple single triangle with no lighting or texturing requiring a couple of hundred lines of code to a complete renderer engine that's able to display realistic human models accurately down to the hairs on their head (requiring thousand or more lines of code with dozens of different shaders and optimisations). What is more, these solutions may be off-line taking minutes or days to calculate or microseconds for real-time interactive virtual environments (video games).

Name: 'VULKAN'
The Vulkan API was a ground-up redesign of the popular OpenGL API, previously referred to as the 'Next Generation OpenGL' (GLNext) initiative - however, over time it was decided to rename the API to 'Vulkan' to help emphasis the radical change in thinking, i.e., the aim to provide applications low-level direct control over processor (GPU/APU/CPU) acceleration for maximized performance and predictability.

1.3 Aim of this Book

This book aims to introduce computer graphics programming in a practical context while addressing a number of crucial questions with regard to 'another' graphical application programming interface (API), for example:

- ✔ What exactly is Computer Graphics and the Vulkan API?
- ✔ Why is understanding the 'differences' between the API important?
- ✔ How do you to get started programming a graphical application with Vulkan?

At the end of this book, you should feel comfortable enough to work with the Vulkan API (i.e., create, customize and generate a variety of simple graphical applications). You should be able to explain the core components of the API, and importantly, why and how they fit together to accomplish the necessary graphical technique [12, 9].

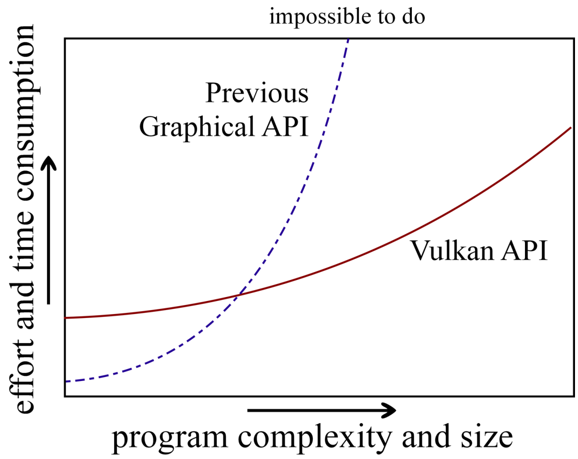

Figure 1.2: Vulkan has a steep learning curve initially - but over time the benefits and freedom provided by the API are rewarded compared to existing solutions (greater optimisations and customisability).

- ✔ Understanding where and why a graphical program 'fails' - e.g., perform worse than current or existing graphical API
- ✔ Dealing with problems, such as, cross-platform, memory leaks, graphical issues, rapid prototyping, versions, ...
- ✔ How to work effectively on complex projects with Vulkan
- ✔ Background introduction to the history of different graphical API
- ✔ Revision on basic graphical principles and techniques (shaders, lighting, transforms, triangles)
- ✔ Managing Vulkan API (structured modular programming)
- ✔ Implement a basic graphical application from the ground up using the native Vulkan API
- ✔ Essential graphical principles and how to implement them with Vulkan
- ✔ How to implement popular graphical effects (e.g., lighting, bump maps, instancing and texturing)

1.4 *Prerequisite (Setting-up Vulkan)*

Pre-requisites to working with Vulkan The computer graphics samples in this book are build around the Vulkan API - hence, to implement and run the examples you'll need to download and install one of the Vulkan SDK libraries on your machine.

To download and install the necessary Vulkan API drivers and SDK (if you don't already have them installed on your system) is very straightforward. For example, a popular Vulkan API SDK is:

Lunar-G (http://lunarg.com/)

In addition, you'll need to have a basic understanding of core programming principles (e.g., functions, pointers, libraries and the ability

Figure 1.3: The LunarG SDK provides the development and runtime components for building, running, and debugging Vulkan applications. This includes the Vulkan loader, Vulkan layers, debugging tools, SPIR-V tools, the Vulkan run time installer, documentation, samples, and demos.

to read simple computer programs written in C, C++ or Java). While basic knowledge of computer graphics concepts would be beneficial (for example, framebuffers and refresh rate), however, it's not required, as you'll be guided through the process of writing a basic graphic applications from the ground-up.

The practical examples and listings in the book are implemented using C/C++.

1.5 Summary

These are exciting times for computer graphics. With advancements in technologies and hardware we're seeing breakthroughs in realism and creativity. The material to create amazing effects is freely available (e.g., free open source libraries, online tutorials and free 3-dimensional models). While computer graphics can seem daunting and difficult initially - especially if your mathematics is a bit rusty - the rewards at the end are well worth the time and effort.

2

Background (OpenGL and Vulkan)

2.1 Introduction

Khronos launched the Vulkan 1.0 specification on February 16th, 2016.

Since OpenGL was first released in 1992 by Silicon Graphics Inc., it has been widely adopted across the world by industry as well as academia. The API reduced the engineering complexities and what followed over the coming years was the birth of visually breathtaking solutions that captured the imagination (both visually and inspirationally). The ability to accomplish stunning computer generated images was made possible through further technological advancements. Computer graphics has become increasingly challenging using conventional approaches and expectations have and continue to grow, especially in areas involved with films, games and virtual reality. One specific challenge is the ability to exploit the advancements in rapidly changing technologies. For example, despite the ready availability of multiple high performance graphics cards, the limitations of existing libraries has made it difficult if not impossible to exploit the full potential of the hardware (distributing the workload for processing and rendering high fidelity images in real-time across multiple devices efficiently [4]). While parallel processing paradigms have become an attractive solution in recent years, with multiple cores and threads working together to offering tremendous performance gains, developing parallel applications that exploit these parallel speed-ups efficiently and reliably is a significant challenge.

Figure 2.1: The Khronos Group is a non-profit, member-funded consortium focused on the creation of royalty-free open standards for parallel computing, graphics and vision processing on a wide variety of platforms and devices. Currently there are 100+ industry-leading company members across the globe.

Vulkan is an exciting multi-platform cross-language graphical and compute interface that exploits the latest 'parallel' hardware architectures. Vulkan provide you and developers with a powerful interface to create stunning visuals for a wide range of applications. Vulkan still follows the same original 'OpenGL' initiatives, i.e., to develop a high quality open source, cross-platform API (Mac, Windows, Linux, Android, Solaris and FreeBSD). OpenGL has come a long way and done amazingly well over the last 25 years (Figure 2.2). Be that as it may, it is time for a major update. As the original OpenGL API follows a state machine architecture this ties the API to a single on-screen context. In addition the OpenGL API is blind to everything the GPU is doing (optimised and managed within the driver - and hidden from the developer). Vulkan takes a different approach - following an object-based API with no global state so all state concepts are localized to a Command-Buffer (you'll learn about Command-Buffers in Section 6.6). What is more Vulkan is more explicit about what the GPU is doing (less hiding what is happening within the driver).

API improvements:

Explicit Control
Multi-Threading Friendly
Direct State Access (DSA)
Bindless Graphics
Framebuffer Memory Info
Texture Barrier
Acceleration for applications (e.g., Browsers, WebGL, ..)

The principle of explicit control, means you promise to tell the driver every detail. So the driver doesn't have to guess or make assumptions. In return, the driver is more streamlined and efficient (does what you asked for when you asked for it quickly). For instance, memory management in Vulkan gives the control to the application (total memory usage is more visible and simplifies operations, such as as for streaming data). Remember, the application is in charge (so doing it correctly is your responsibility).

While the latest OpenGL graphical API (known as Vulkan) might seem like another iteration, it is well worth learning or even reviewing. At the same time, Vulkan is in its first release (revision 1.0) - and possesses a huge number of changes/improvements compared to any previous update. Importantly, these improvements should not be ignored, as they offer possibilities that were previously not feasible. These key features will help you get more out of GPUs. However, to gain improvements it is important you understand the differences (i.e., applications need to be written differently to utilize these additional features and control - OpenGL $!=$ Vulkan). As shown in Figure 2.3, you'll notice the shift of power between the driver and the application. Vulkan's abstraction means your application is much closer the hardware compared to traditional APIs. Your application is driving the hardware directly, while leaving just enough abstraction to make things portable. You're not being second-guessed by the driver, while at the same time you're not being first-guessed either. You now have

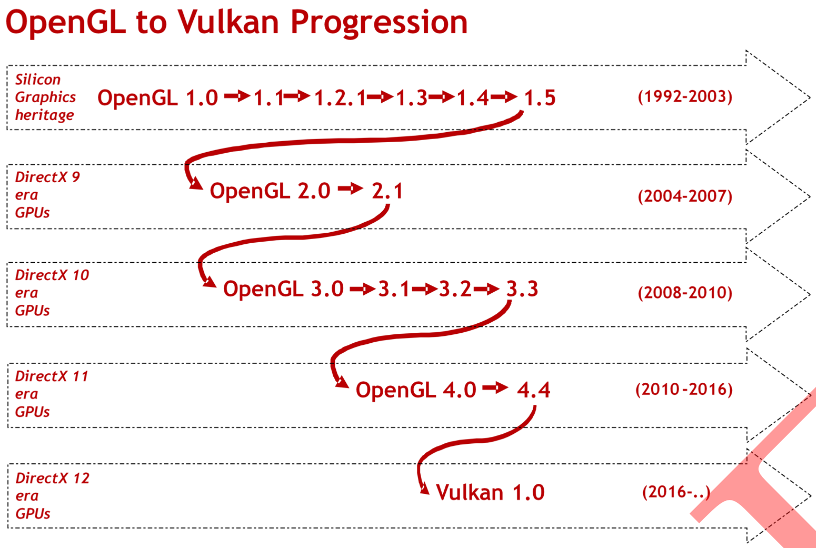

Figure 2.2: Evolution of the OpenGL API to the most recent incarnation known as 'Vulkan'.

all the control you need to get the best out of your hardware. If it doesn't go fast in Vulkan, it's your fault (of course, remember, with great power comes great responsibility).

A few of the "big tick" items with Vulkan is:

- Explicit control,
- Support for multi-core/threading,
- Predictability,
- Texture formats, memory management, and syncing are client-controlled
- Vulkan drivers do no error checking and
- Bandwidth efficiency.

2.2 *History of Vulkan*

The Vulkan API was designed and is maintained by the Khronos Group to meet current and future demands for achieving high performance rendering and compute solutions. The Vulkan API achieves this by allowing greater low level control (explicitly) - moving away from 'default' parameters/assumptions set within the driver. The developer has to manage the memory, resource updates, batching, scheduling, ... Hence, the Vulkan API initially seems verbose and complicated due to the large amount of initiation and management (through functions, parameters and structures), yet this is crucial for Vulkan's success. It should also be noted, that DirectX 12 from Microsoft follows a similar design to Vulkan (explicit low level control). For instance, previously,

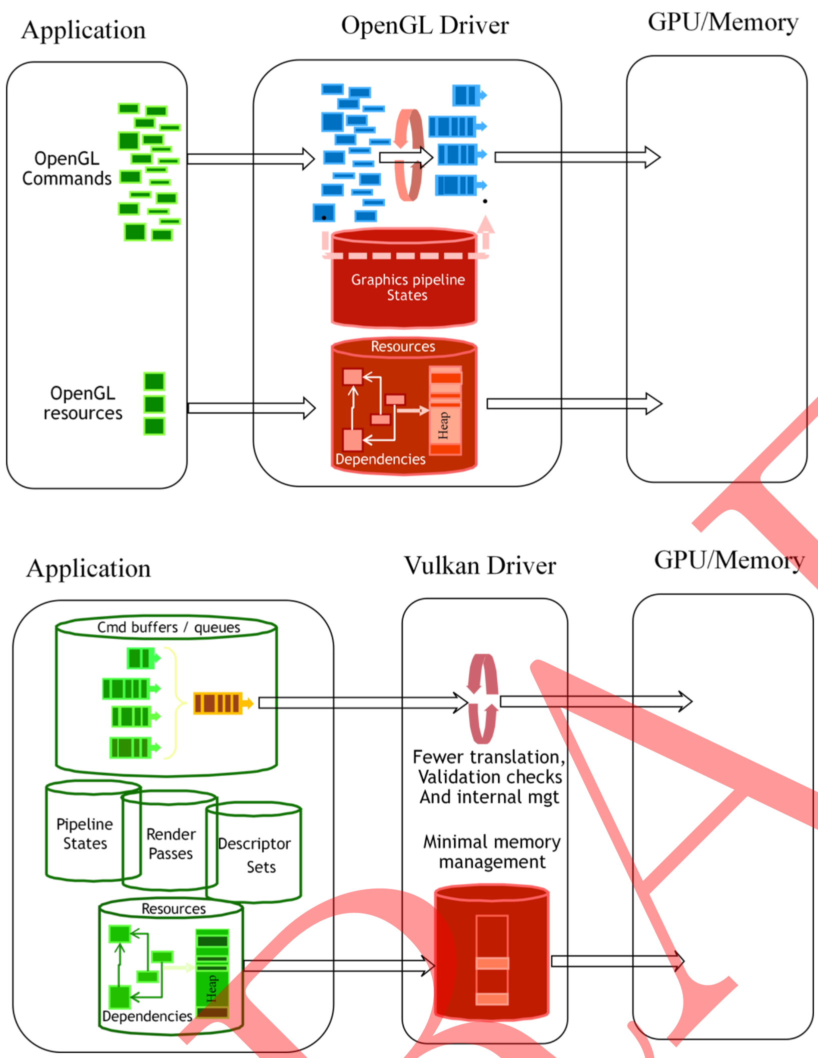

Figure 2.3: High-level view of what has changed between OpenGL and Vulkan. Importantly, the shift in power and work from the driver to the application. The application is now responsible for a number of crucial tasks that were previous hidden to the developer, such as, memory management, resources and command-buffers. This modifications provides a more 'streamline' and efficient solution (bringing the application developer nearer to the hardware).

'OpenGL' did not address multi-threading and was not designed to support the concurrent and parallel paradigm which would be a serious problem in todays multi-core multi-threaded environment. However, the Vulkan API is designed to exploit these multi-threaded environments (and is how it is able to outperforms previous API).

2.3 *11 Steps*

You'll see an overview of the essential components in most Vulkan graphical applications in Figure 2.4. To complement this programming section, the components have been grouped into 11 distinct steps. From step 1 which initializes the application and creates the window -

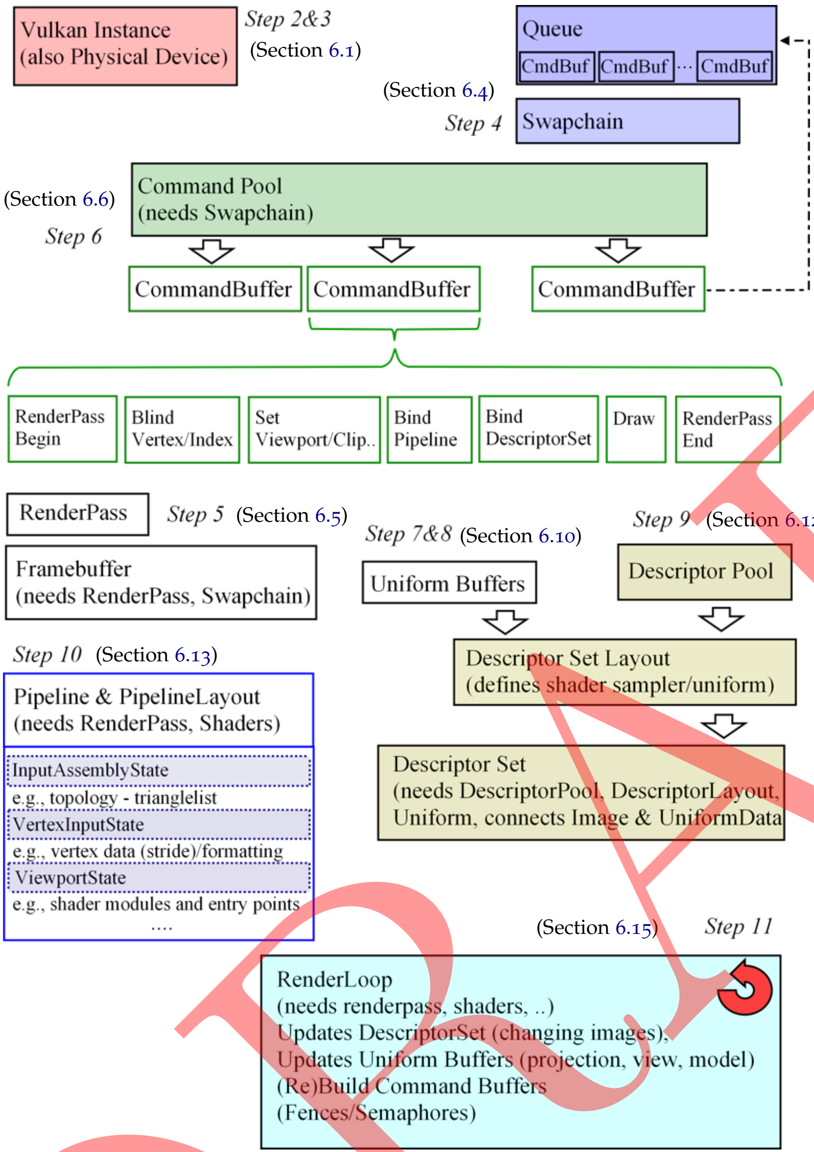

Figure 2.4: Decomposition of the most popular Vulkan components available in common graphical applications. The illustration provides an overview of the elements and how they fit together in a simple application. You'll review and discuss each of the components in the following sections (i.e., from initializing Vulkan and the physical device to building a Command-Buffer and rendering to the screen).

to the final stage 11 which performs the updates and rendering. Each step provides a self-contained set of material which is used to breakup an otherwise complex system. The steps enable you to progress in an orderly manner as you learn the rational behind each of the elements and how they fit together (e.g., swap-chains and command-buffer). Each step builds upon the previous steps and enable you to incrementally build a complete graphical application using the Vulkan API that utilizes all of the features (in a modular manner).

Briefly, the 11 steps are:

1. Initialize application and create a window (operating system specific)
2. Initialize Vulkan (Vulkan Instance)
3. Initialize Device (e.g., GPU)
4. Create Swap-Chain (managing the display output)
5. FrameBuffer & Render-Pass (output image surfaces)
6. Command-Buffer & Command-Pool (essential for graphics - as all draw commands need to be in a command-buffer)
7. Vertex Data (geometry you'll be drawing)
8. Shaders & Uniform Buffers (essential for graphics to have a vertex and fragement shader in addition to any parameters/passing of data to the shaders)
9. Descriptors (glue that holds everything together, such as, the shaders and geometry vertex data)
10. Graphics Pipeline (connecting everything together and enabling features)
11. Render Loop (drawing/syncing)

2.4 *Naming Convention*

The Vulkan API variables and functions follow a consistent naming convention. While both variables and functions start with the letters 'vk', you need to remember, functions start with a lowercase letter while variables start with an uppercase letter, for example:

Function: **vkCreateInstance(..)**

Variable: **VkResult**

In the example listings that follow in subsequent sections, the Vulkan API functions and structures have been emphasised to help you identify the key elements.

2.5 *Exercises*

2.5.1 *Chapter Questions*

Question When was the Vulkan 1.0 specification released?

Question What is the naming convention for Vulkan variables and functions?

Question What is the root methodology behind Vulkan compared to previous graphical API?

Question What is a ray tracing algorithm and how does it compare to a rasterization approach?

3
Mathematics

3.1 Introduction

There are a few fundamental mathematical concepts that are indispensable when working with computer graphics and geometric systems (e.g, vectors and matrices including concepts such as normals and the dot product). The main mathematical tools that you'll review in this chapter are:

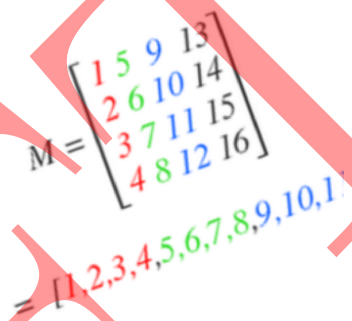

Figure 3.1: A large majority of computer graphics principles requires you understand common mathematical topics (e.g., matrix and vector mathematics, linear algebra and trigonometry).

- Vectors
  - Dot
  - Cross
- Matrices
  - Transforms
- Quaternions
  - Rotations

Hence, you'll briefly review the workings and implementation details for each mathematical concept. However, in practice you may prefer to use existing pre-written libraries (e.g., glm), but be careful you don't get caught with problems, such as, "handed" convention (i.e., left or right handed differences) or function speed-up hacks, which can cause large numerical errors.

3.2 Vector

3.2.1 What is a Vector?

A vector represents a mathematical or physical direction and length (or magnitude) and is depicted by an arrow (with the arrow symbolizing the direction and the length of the arrow the magnitude). For example, the wind has a direction and speed, as shown on weather maps. You can have different dimensions of vectors (i.e., 1D, 2D, 3D, 4D, ...). Note, a 1D vector would just be a scalar float. However, you'll primarily be dealing with 3D vectors composed of an x, y, and z). If you want a 2D vector just remove the z. In code, a vector is nothing more than an array of variables (e.g., float[3]). So that you can distinguish the dimensions of your vector, you'll add the number to the end, e.g., "Vector3" and "Vector2". You'll use a class or structure to represent your vector since it makes the code more readable and you'll be able to exploit operator overload.

Let's get this out the way right at the start - what is the difference "in code" between a "Point" and a "Vector"? For example, a Vector3 and a Point3 structure. The answer: Nothing! The code is identical, except for the name of course.

In short, don't make work for yourself. Don't create structures or variables that accomplish the same task but use different names. For example, you might be tempted to use Vectors for direction and Points for position. However, the name of the variable should be sufficient for a detailed description of what the variable is does. For example, Listing 3.1 shown below:

Listing 3.1: Application of Vector3 (e.g., Positions and Directions)

```
Vector3 position;
Vector3 direction;
Vector3 velocity;
Vector3 force;
```

3.2.2 *Vectors and Points*

A 3D vector differ from a 3D point tuple (x,y,z) in 3D game mathematics. They are different 'mathematically', while you represent them the same pragmatically. The difference is that a vector is an algebraic object that may or may not be given as a set of coordinates in some space. A *point is just a point* given by coordinates. Generally, you can conflate the two. An intuitive way to think about the association between a vector and a point is that a vector tells you how to get from the origin (that one point in space to which you assign the coordinates $(< 0, 0, 0 >)$ to its associated point. While in code they may appear the same (e.g., Vector3 for a variable), ensure you know 'mathematically', what that variable stands for (i.e., a 3d position in space or a vector direction with magnitude).

3.2.3 *Vector3*

Listing 3.2: Unsophisticated Vector3 Class Implementation

```
class Vector3
{
public:
  float x;
  float y;
  float z;
};
```

Without vectors, basic geometric calculations would be very complex, difficult to read, and time consuming when debugging. Furthermore, once you understand vectors and how to use them, in combination with the various routines (e.g., dot and cross product) you'll be able to tackle daunting geometric problems without even breaking a sweat.

3.2.4 *Dot Product*

In a nutshell, the dot product is amazing. It's flexible, computationally efficient, and straightforward to use. To summarize, here are the main features the dot product offers:

- Magnitude squared distance of two vectors is the dot product operation
- Sign of the result of the dot product enables us to determine if vectors are facing towards or away from one-another

  Word of caution, this operation does not require the vectors to be of unit-length, so you can avoid the cost of normalizing the vectors
- Cosine of the angle between two vectors

  Warning, the vectors must be of unit-length, also the 'sign' of direction is not provided (i.e., only provides the shortest path and doesn't tell us the direction)
- Project a vector onto another vector

  Note, the vector you are projecting onto should be a unit-vector
- Dot product doesn't involve any complex computational operations (e.g., sqrt, sin) and can be performed using simple multiplication and addition

  The dot product can be speeded-up on modern hardware technology since operations such as multiplication can be performed in parallel (e.g., dot product can be done in a single instruction on some processors)

The dot product returns a single scalar value and can easily be implement, as shown in Listing 3.3.

Listing 3.3: Unsophisticated Vector3 Dot Product Implementation.

```
inline
float Dot( const Vector3& A, const Vector3& B)
{
  return ( A.x * B.x + A.y * B.y + A.z * B.z );
```

```
};
```

3.2.5 *Cross Product*

While the dot product may come first for usefulness and features the cross product is not far behind for providing a similar list of useful operations. The cross product of two vectors (a and b) is written as $a \times b$ and returns a vector. In three dimensional space, the cross product of two vectors is a vector that is "perpendicular" to both the initial vectors.

The main features of the cross product are:

- Calculates a vector perpendicular to two unit vectors
- Can be combined with the dot product to provide a direction of rotation between two unit vectors (i.e., dot product provides the angle between the two unit vectors but doesn't provide the direction of rotation)
- Cross product doesn't involve any complex computational operations (e.g., sqrt, sin) and can be performed using simple multiplication, addition and subtraction

  Note, modern hardware can perform the cross product in a single operation due to the parallel nature of the operation
- the area of a parallelogram with sides AB and AC is equal to the magnitude of the cross product of vectors representing two adjacent sides (while the area of a triangle would be half that)

The direction of the resulting vector cross product is given by the "right-hand" convention. With your right hand, if your first finger is vector a, and your second finger is vector b, then your thumb is the cross product result $a \times b$. The implementation details in code are shown in Listing 3.4.

Listing 3.4: Unsophisticated Vector3 Cross Product Implementation.

```
inline
Vector3 Cross( const Vector3& A, const Vector3& B)
{
    Vector3 vec;
    vec.x =  (A.y*B.z) - (B.y*A.z);
    vec.y =  (A.z*B.x) - (B.z*A.x);
    vec.z =  (A.x*B.y) - (A.y*B.x);
    return vec;
};
```

Be warned that the cross product is "non-commutative", i.e., $(a \times b)$ does "not" equal $(b \times a)$.

3.2.6 *Reconstructing Angles from Positions*

Given a set of points, you can reconstruct the link's angle from the positional information as shown in Figure 3.2. This can be valuable when you have a set of animation capture points, and you want to reconstruct the articulated character's bone structure (i.e., rigid bodies and joint angles).

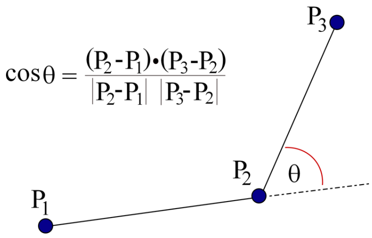

Figure 3.2: **Direction to Angle** - Illustrating how to reconstruct angles from points. You subtract P_3 and P_1 from P_2 to construct two vector directions. Dividing them by their magnitudes normalizes them (i.e., to their unit length values). Finally, the dot product of the two vectors gives us the cosine of the angle between them.

3.2.7 *Plane Equation*

The plane equation is a mathematical method for representing the valuable concept of a planar surface. The plane equation is probably one of the most useful tools in your algorithm artillery. It boasts the advantage of being uncomplicated and computationally fast. To start with, you can define a plane mathematically by four different methods, but you most commonly represented it as 'a point and a normalized vector'. The normalized vector is perpendicular to the plane, while the known point can be anywhere on the planes surface. As you'll see, the Cartesian form of the plane equation is formally defined as: $A\ x + B\ y + C\ z + d = 0$, where $< A, B, C >$ is the vector normal to the plane, $< x, y, z >$ is a point on the plane, and d is the shortest distance from the plane to the origin. The plane equation is used for an assortment of crucial techniques and forms the backbone of a number of fundamental algorithms.

Plane Equation & Dot Product The plane equation can be calculated using the dot product. To define a plane, you need two pieces of information. First, you need a point on the plane, anywhere on the plane; it doesn't matter as long as the point is on the plane. Second, you need the normal of the plane (i.e. the direction the plane is facing).

$$d = \hat{n} \cdot \vec{p} \tag{3.1}$$

where $\hat{n}$ is the plane normal in Cartesian coordinates (unit-length), while the $\vec{p}$ represents the coordinates of a point on the plane, and d

represents the shortest distance from the plane to the origin. Note, the point $\vec{p}$ can be any point on the surface of the plane.

3.2.8 *Support Function*

Many algorithms make use of a mathematical tool called the **support function**, a.k.a **support mapping**. A support function takes a **direction** and an **array of vertices** as input and returns a point as output. The **output point is the furthest point along the given direction** given all the vertices. Note, there can be multiple points that are valid support function outputs for a particular array of vertices. For instance, the support function of an AABB, given the positive x-axis direction, can return any point on the AABB's face in the positive x-axis direction.

3.3 *Matrix*

3.3.1 *Why Matrices?*

Matrices are a compact way of representing and combining transformations (e.g., rotations and translation). Matrices are so common that most computer hardware (e.g., graphical processing units (GPUs) and CPUs) are optimized to perform very efficient matrix operations (i.e., with special instructions and by means of parallelization).

3.3.2 *Column or Row Major*

A matrix can be ordered using either Column or Row ordering (i.e., depending upon your preference). While DirectX uses Row Major ordering to store the matrix in memory, OpenGL uses Column Major ordering. For this book, you primarily use Column Major ordering.

$$\underset{\textit{row matrix}}{M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}} \qquad \underset{\textit{column matrix}}{M = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}}$$

Figure 3.3: **Column or Row Major** - Visually illustrating the difference between a column and row matrix organisation.

```
/*
* Column-major 4x4 matrix
*
* Layout:
*     0  4  8  12
*     1  5  9  13
*     2  6  10 14
*     3  7  11 15
*
```

```
 *  3x3 Rotation Matrix Indices
 *     0  4  8
 *     1  5  9
 *     2  6  10
 *
 *  3x1 Translation Indices
 *     12
 *     13
 *     14
 *
 */
```

$$M = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

$$float\ \ m[16]\ =\ [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]$$

Figure 3.4: **Matrix Elements** - The OpenGL specification for a 4x4 matrix uses column-major ordering. This means that the values of the matrix are stored by filling the columns with values, and only moves onto the next column once the current column is completely filled. This column-major order is adhered to throughout all the matrix operations within OpenGL. Remember, every fourth consecutive elements in an array represents a column in a 4ÃŮ4 matrix.

3.3.3 *A 4x4 Matrix*

A 4x4 matrix (aka a homogeneous transformation matrix) can contain multiple different transformations (e.g., scaling, rotation, and translation), as shown in Figure 3.5. Rather than working with multiple different types of matrix, you will only work with a 4x4 matrix. Note, just in-case you didn't catch-on, a vector3 is technically a 1x3 matrix.

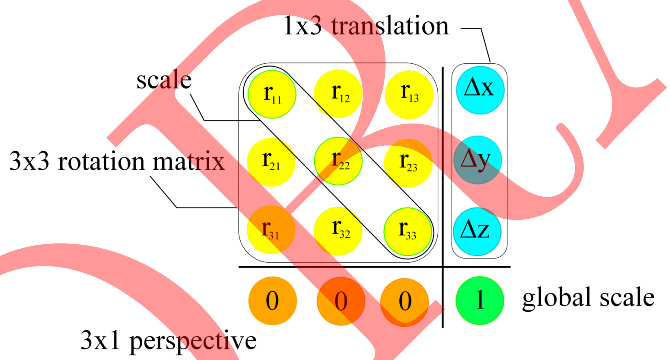

Figure 3.5: **4x4 Homogeneous Transformation Matrix** - Illustrating the different parts of a 4x4 matrix that represent the different transformations.

Figure 3.5 illustrates the decomposition of a 4x4 matrix into a 3x3 rotation matrix and a 1x3 translation matrix. Also, it shows how the diagonal components of the 3x3 rotation affect scaling along the x, y, and z axis., while the global scaling and perspective values are typically fixed.

3.3.4 *Creating a Matrix*

A matrix is just an array of variables. For example, an uncomplicated 4*x*4 matrix is shown below in Listing 3.5 is merely an array of 16 floats.

Listing 3.5: Uncomplicated Matrix.

```
class Matrix4
{
public:
  float M[16];
};
```

For C++ and C#, you take advantage of accessor functions and operator overloading to make using variables easier and safer (i.e., sanity checks within the accessors), as shown in Listing 3.6.

Listing 3.6: Basic Matrix4 class for C++.

```
class Matrix4
{
public:
  // Row - Column Format
  // e.g., mat.Get(2,4) is row=2, and column=4
  // or mat(2,4) - using operator overloading
  float M[16];

  // Accessor with sanity checks (i.e., boundary and
  // valid number asserts)

  float Get(int row, int col) const
  {
    DBG_ASSERT(row>=0 && row<4);
    DBG_ASSERT(col>=0 && col<4);
    return M[row*4+col];
  }// End Get(..)

  // Note - you can't overload operator[] to
  // accept multiple arguments. Instead -
  // instead you can overload operator() if you want to access
  // values using (x,y) syntax
  float& operator() (int row, int col)
  {
    DBG_ASSERT(row>=0 && row<4);
    DBG_ASSERT(col>=0 && col<4);
    return M[row*4+col];
  }
};

// example:
Matrix4 mat;
mat(0,3) = 2;
// and
float val = mat.Get(0,3);
```

Note, a matrix can be stored in row-column or column-row form. Make sure you know which is which and be consistent.

3.3.4 *Identity Matrix*

The identity matrix is analogous to the number 1. If you multiply any matrix by an identity matrix, you will get the original matrix. The format for an identity matrix is all zeros except for the diagonal components, as shown in Equation 3.2.

$$M_{identity} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

The implementation for creating an identity matrix is shown in Listing 3.7.

Listing 3.7: Creating a 4x4 Identity Matrix.

```
// Returns an instance of an identity matrix
static
Matrix4 Identity()
{
  Matrix4 m;
  m(0,0)=1;  m(0,1)=0;  m(0,2)=0;  m(0,3)=0;
  m(1,0)=0;  m(1,1)=1;  m(1,2)=0;  m(1,3)=0;
  m(2,0)=0;  m(2,1)=0;  m(2,2)=1;  m(2,3)=0;
  m(3,0)=0;  m(3,1)=0;  m(3,2)=0;  m(3,3)=1;
  return m;
}
```

3.3.4 *Translation Matrix*

The translation matrix represents a 3D world positions (i.e., an x, y, and z Cartesian point in space).

Essentially, if you start with an identity matrix, which does nothing when multiplied with another matrix. Then the bottom three values describe the translational information, as shown in Equation 3.3.

$$M_{translation} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

Listing 3.8: Creating 4x4 Translation Matrix.

```
static
Matrix4 CreateTranslation(float x, float y, float z)
{
  Matrix4 m = Matrix4.Identity;
```

```
  m(0,3)=x;
  m(1,3)=y;
  m(2,3)=z;
  return m;
}
```

3.3.4 *Scale Matrix*

You'll want to make things smaller and bigger! You can scale objects with a scaling matrix. You can scale the x, y, and z axis by modifying the diagonal elements of the matrix, as shown in Equation 3.4.

$$M_{scale} = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

Listing 3.9: Creating 4x4 Scale Matrix.

```
Matrix4 CreateScale(float x, float y, float z)
{
  Matrix4 m = Matrix4.Identity;
  m(0,0)=x;
  m(1,1)=y;
  m(2,2)=z;
  return m;
}
```

3.3.4 *Rotation Matrix*

You need to be able to rotate your objects. You formulate the three main axis rotation matrices, as shown in Equation 3.5.

$$\begin{aligned} M_{XRotation} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(-XAngle) & -sin(-XAngle) & 0 \\ 0 & sin(-XAngle) & cos(-XAngle) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ M_{YRotation} &= \begin{bmatrix} cos(-YAngle) & 0 & sin(-YAngle) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(-YAngle) & 0 & cos(-YAngle) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ M_{ZRotation} &= \begin{bmatrix} cos(-ZAngle) & -sin(-ZAngle) & 0 & 0 \\ sin(-ZAngle) & cos(-ZAngle) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{3.5}$$

The order that you multiply the matrices determines the order the rotations will be applied to the point. For example:

$$
\begin{aligned}
&P \times (X \times Y \times Z) \text{ Rotates in X, Y, then Z}\\
&P \times (Y \times X \times Z) \text{ Rotates in Y, X, then Z}\\
&P \times (Z \times X \times Y) \text{ Rotates in Z, X, then Y}
\end{aligned}
$$

where P is the point, and X, Y, and Z represent the matrix-axis rotation.

Listing 3.10: Rotation Matrix Implementation.

```
static
Matrix4 CreateRotationX(float ax)
{
  Matrix4 m = Matrix4.Identity();
  m(1,1) = (float)Math.Cos(-ax);  m(1,2) = -(float)Math.Sin(-ax);
  m(2,1) = (float)Math.Sin(-ax);  m(2,2) =  (float)Math.Cos(-ax);
  return m;
}

Matrix4 CreateRotationY(float ay)
{
  Matrix4 m = Matrix4.Identity();
  m(0,0) =  (float)Math.Cos(-ay);  m(0,2) = (float)Math.Sin(-ay);
  m(2,0) = -(float)Math.Sin(-ay);  m(2,2) = (float)Math.Cos(-ay);
  return m;
}

Matrix4 CreateRotationZ(float az)
{
  Matrix4 m = Matrix4.Identity();
  m(0,0) = (float)Math.Cos(-az);  m(0,1) = -(float)Math.Sin(-az);
  m(1,0) = (float)Math.Sin(-az);  m(1,1) =  (float)Math.Cos(-az);
  return m;
}
```

3.3.5 *Matrix-Matrix Multiplication*

You can construct matrices that represent different transformations (e.g., scaling, translation, and rotation), which you combine through multiplication.

Always remember matrix multiplication is **NOT commutative**. For example, if you want to rotate the object first then translate its position, you have to be sure you do the multiplication in the correct order; otherwise, you'll end up, translating the object then rotating it.

Listing 3.11: Matrix Multiplication Implementation (result = A * B)

```

Matrix4 Multiply(const Matrix4& ma, const Matrix4& mb)
{
  Matrix4 result;;
```

```
  for ( int i = 0; i < 4; ++i )
    for ( int j = 0; j < 4; ++j )
      result(i,j) =   ma.Get(i,0) * mb.Get(0,j)
                    + ma.Get(i,1) * mb.Get(1,j)
                    + ma.Get(i,2) * mb.Get(2,j)
                    + ma.Get(i,3) * mb.Get(3,j);
  return result;
}
```

3.3.6 'Pure' Rotation

Matrices that contain only rotation possess special features. For example, they can be easily inverted and converted to and from quaternion or axis-angle format.

3.3.6 Orthogonal Matrices (Useful-Axis)

A matrix that contains only rotational information is termed an 'orthogonal' matrix.

3.3.6 Transpose and Inverse

The inverse of an orthogonal (i.e., 'pure' rotation) matrix is its transpose (i.e., you swap the columns and rows). This is extremely valuable since it is computationally fast, since it requires no complex mathematical operations (e.g., sin and cos), and is straightforward and simple to implement in code, as shown in Listing 3.12.

Listing 3.12: Matrix Transpose Implementation.

```

Matrix4 Transpose(const Matrix4& m)
{
  Matrix4 result;
  for ( int i = 0; i < 4; ++i )
    for ( int j = 0; j < 4; ++j )
      result(i,j) = m.Get(j,i);
  return result;
}
```

3.3.7 Transforming a Vector

A vector is basically a matrix with a single row, or column, depending upon your configuration. You can multiply your 4x4 matrix by a 4x1 vector (you'll convert your 3x1 to a 4x1 vector with the last component set to zero). The operation is shown in Equation 3.6.

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} * \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} = \begin{bmatrix} a & b & c & d \end{bmatrix} \tag{3.6}$$

Listing 3.13: Matrix-Vector Transform Implementation.

```
// Row-Vector Convention
Vector3 Transform (const Vector3& v, const Matrix4& m)
{
    float result[4];
    for ( int i = 0; i < 4; ++i )
    {
      result[i] = v.X*m(0,i) + v.Y*m(1,i) + v.Z*m(2,i) + m(3,i);
    }
    return Vector3( result[0]/result[3],
                    result[1]/result[3],
                    result[2]/result[3] );
}
```

3.3.7 *Little Test*

So does your implementation work? You'll do a simple example to demonstrate your matrix and vector are performing the correct calculation. Don't just walk away and 'assume' it works. You should always ask the question, have I tested and am able to 'prove' that the code works - even if it's just modifying a few lines for optimisation reasons - did the optimisation or modification break the original implementation?

Let's create a simple Vector3 (e.g., 0,1,0) pointing straight-up, then you'll create a simple rotation matrix (e.g., rotate $\frac{\pi}{2}$ (i.e., 90 degrees) around z-axis). If you typed the code correctly, you should end up with a Vector3 pointing to the right (e.g., -1,0,0). Listing 3.14 demonstrates a simple implementation example for transforming a vector in code.

Note!!! You "Always" work with radians!! Not degrees, potatoes, or bananas, but "radians". Furthermore, positive rotation is counterclockwise, not clockwise. That is why when you rotate the Vector3(0,1,0), around the z-axis by $\frac{\pi}{2}$, you get Vector3(-1,0,0).

Listing 3.14: Basic Matrix-Vector Transform Sanity Test.

```

```

```
// Start with <0,1,0>, rotate it, and get <-1,0,0> back
Vector3 vy   = Vector3(0,1,0);
Matrix4 rotZ = Matrix4.CreateRotationZ( (float)Math.PI*0.5f );
Vector3 vr   = Vector3.Transform( vy, rotZ );
// If all went well, vr equals (-1,0,0); well approximately, e.g., (-0.9999, 0, 0),
      due to numerical errors and floating point precision ;)
```

3.3.8 *Matrix Inversion*

A matrix is just a rectangle array of numbers or symbols organised into rows and columns. For example:

$$
\begin{aligned}
[A] &= \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \\
[A] &= \begin{bmatrix} 0.25 & 0.33 \\ 0.125 & 0.66 \end{bmatrix}
\end{aligned}
\tag{3.7}
$$

So given the popular equation $F = ma$, if you know the force and the acceleration, you can work out the mass from $m = \frac{F}{a}$. However, for matrices, the division operation does not exist, hence, you use the 'inverse': $m = a^{-1}F$.

Matrix Inverse Properties Given a square matrix $[A]$ (i.e., equal number of rows and columns), then you can say:

$$[A]^{-1}[A] = [A][A]^{-1} = [I] \tag{3.8}$$

where $[I]$ is the identity matrix (i.e., matrix equivalent of 1).

There are two methods for inverting a matrix:

- Analytical
- Numerical

Analytical Matrix Inversion For small matrix problems (e.g., 2x2 or 3x3), the solution can be computed by hand, for example:

$$
\begin{aligned}
[A] &= \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \\
[A]^{-1} &= \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} \\
[A]^{-1}[A] &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = [I]
\end{aligned}
\tag{3.9}
$$

Numerical Matrix Inversion When an analytical solution does not exist, then a numerical solution can be sought. For example:

$$
\begin{aligned}
[A] &= \begin{bmatrix} 0.25 & 0.33 \\ 0.125 & 0.666 \end{bmatrix} \\
[A]^{-1} &= \begin{bmatrix} 5.31734 & -2.6347 \\ -0.998 & 1.996 \end{bmatrix} \\
[A]^{-1}[A] &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = [I]
\end{aligned}
\tag{3.10}
$$

Techniques for numerically inverting a matrix, include:

- Gaussian Elimination (LU factorization, Gauss-Seidel)
- Singular Value Decomposition (SVD)
- Cholesky Factorization (symmetric defined matrices)

When considering a numerical routine, computational cost and robustness are important factors - for example, you may want the algorithm to converge on a best guess solution for singular matrix problems (i.e, non-convertible matrix - analogous to a divide by zero issue).

Singular Systems If a matrix is 'not' invertible it is said to be singular (it exists on its own). When a matrix is singular, the determinant of a matrix is equal to zero.

Singular systems arise when:

- the equations representing the rows in a matrix are closely inter-related
- data in the matrix contains significant errors which makes it seem as if the rows in the matrix are closely inter-related

Determinant The determinant of a matrix is a single scalar value. Every square matrix has a determinant. For example, to calculate the determinant for a 2×2 matrix:

$$
\begin{aligned}
det[A] &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \\
&= ad - bc
\end{aligned}
\tag{3.11}
$$

When the determinant of a matrix is zero, it is not invertible.

3.4 *Quaternion*

Quaternions are an efficient, straightforward and robust way of representing rotations. You can represent a rotation using a 3x3 matrix, however, a quaternion only uses 4 variables instead of the 9 variables for a 3x3 matrix. Allows you to easily be interpolated, combine, and re-normalized orientations during drifting (numerical errors).

3.4.1 *Why Quaternions?*

If quaternions are compared with other types of methods for representing rotation (e.g., Euler's angles, matrices, axis-angle) the quaternion comes out on top. In summary:

- They don't suffer from gimbals lock
- They use the minimum number of variables (i.e., 4-floats) to *uniquely* represent a rotation with no ambiguity
- They are easy to combine (i.e. through multiplication the same as with matrices)
- They can be inversed easily (i.e., unit-quaternion's inverse is its conjugate, which is simply the negative of the vector components)

  Hence, you can calculate angular difference between pairs of unit-quaternions easily and fastly
- Interpolating is a breeze
- Drifting due to numerical errors is easier to correct (i.e., re-normalizing the unit-quaternion) compared to matrices

3.4.2 *Unit-Quaternion (Always)*

In the majority of cases your quaternions will always be unit-quaternions. If they aren't then something has gone wrong. Hence, assert and check that the length of your quaternions is always (approximately) equal to one.

3.4.3 *Creating a Quaternion*

Essentially, a quaternion is just a 4 vector class, and its implementation is very simple, as shown in Listing 3.15. However, it's all the helper methods that make the quaternion tool invaluable (e.g., multiplication and interpolation methods) that you go into next.

Listing 3.15: Implementation of a Quaternion class.

```
class
Quaternion
```

```
{
public:
  float w, x, y, z;
};
```

3.4.3 *Quaternion from Axis-Angle*

Listing 3.16: Quaternion From Axis-Angle Implementation.

```

Quaternion QuaternionFromAxisAngle(const Vector3& axis, float angle)
{
  Quaternion q;
  q.X = axis.X * (float)Math.Sin(angle/2);
  q.Y = axis.Y * (float)Math.Sin(angle/2);
  q.Z = axis.Z * (float)Math.Sin(angle/2);
  q.W = (float)Math.Cos(angle/2);
  return q;
}
```

3.4.3 *Quaternion to Axis-Angle*

Listing 3.17: Quaternion To Axis-Angle Implementation.

```

void QuaternionToAxisAngle(const Quaternion& q,
               Vector3& outAxis,
               float&   outAngle)
{
  outAngle = 2 * (float)Math.Acos(q.w);
  float s = (float)Math.Sqrt(1-q.w*q.w); // assuming quaternion normalised then w is
      less than 1, so term always positive.
  if (s < 0.001)
  { // test to avoid divide by zero, s is always positive due to sqrt
    // if s close to zero then direction of axis not important
    outAxis.x = q.x; // if it is important that axis is normalised then replace with
        x=1; y=z=0;
    axis.y = q.y;
    axis.z = q.z;
    return;
  }

  outAxis.X = q.x / s; // normalize axis
  outAxis.Y = q.y / s;
  outAxis.Z = q.z / s;
}
```

3.4.3 *Quaternion to Matrix*

The top left 3x3 part of the rotation matrix is formed with Equation 3.12.

$$\begin{bmatrix} 1-2q_y^2-2q_z^2 & 2q_xq_y-2q_zq_w & 2q_xq_z+2q_yq_w & 0 \\ 2q_xq_y+2q_zq_w & 1-2q_x^2-2q_z^2 & 2q_yq_z-2q_xq_w & 0 \\ 2q_xq_z-2q_yq_w & 2q_yq_z+2q_xq_w & 1-2q_x^2-2q_y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.12}$$

Listing 3.18: Quaternion to Matrix Implementation.

```

Matrix4 QuaternionToMatrix(const Quaternion& q)
{
  float sqw = q.W*q.W;
  float sqx = q.X*q.X;
  float sqy = q.Y*q.Y;
  float sqz = q.Z*q.Z;

  Matrix4 m = Matrix4.Identity();

  // invs (inverse square length) is only required if quaternion is not already normalised
  float invs = 1 / (sqx + sqy + sqz + sqw);
  m(0,0) = ( sqx - sqy - sqz + sqw)*invs; // since sqw + sqx + sqy + sqz =1/invs*invs
  m(1,1) = (-sqx + sqy - sqz + sqw)*invs;
  m(2,2) = (-sqx - sqy + sqz + sqw)*invs;

  float tmp1 = q.X*q.Y;
  float tmp2 = q.Z*q.W;
  m(1,0) = 2.0f * (tmp1 + tmp2)*invs;
  m(0,1) = 2.0f * (tmp1 - tmp2)*invs;

  tmp1 = q.X*q.Z;
  tmp2 = q.Y*q.W;
  m(2,0) = 2.0f * (tmp1 - tmp2)*invs;
  m(0,2) = 2.0f * (tmp1 + tmp2)*invs;
  tmp1 = q.Y*q.Z;
  tmp2 = q.X*q.W;
  m(2,1) = 2.0f * (tmp1 + tmp2)*invs;
  m(1,2) = 2.0f * (tmp1 - tmp2)*invs;
  return m;
}
```

3.4.3 *Quaternion from Matrix*

As show in Equation 3.12, with the rule that your quaternion is a unit-quaternion (i.e., $q = (q_w, q_x, q_y, q_z)$ where $|q| = 1$)

You need to know how a rotation matrix (i.e., a 'pure' 3x3 rotation matrix without scaling) can be compared with the result of a quaternion (e.g., see Figure 3.5 for the components of a matrix.).

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} q_w^2 + q_x^2 - q_y^2 - q_z^2 & 2(q_x q_y - q_z q_w) & 2(q_x q_z + q_y q_w) \\ 2(q_x q_y + q_z q_w) & q_w^2 - q_x^2 + q_y^2 - q_z^2 & 2(q_y q_z - q_x q_w) \\ 2(q_x q_z - q_y q_w) & 2(q_y q_z + q_x q_w) & q_w^2 - q_x^2 - q_y^2 + q_z^2 \end{bmatrix} \tag{3.13}$$

by remembering that $q_w^2 + q_x^2 + q_y^2 + q_z^2 = 1$, you can rearrange and solve Equation 3.13 to calculate the 3x3 rotation matrix components.

Listing 3.19: Quaternion from Matrix.

```

static float SIGN(float x) {return (x >= 0.0f) ? +1.0f : -1.0f;}
static float NORM(float a, float b, float c, float d) {return sqrt(a * a + b * b + c
    * c + d * d);}

static
Quaternion QuaternionFromMatrix(const Matrix4& m)
{
  /*
        | 00, 01, 02 |
    m = | 10, 11, 12 |
        | 20, 21, 22 |

    q = | qx, qy, qz, qw |
  */
  float qx = ( m(0,0) + m(1,1) + m(2,2) + 1.0f) / 4.0f;
  float qy = ( m(0,0) - m(1,1) - m(2,2) + 1.0f) / 4.0f;
  float qz = (-m(0,0) + m(1,1) - m(2,2) + 1.0f) / 4.0f;
  float qw = (-m(0,0) - m(1,1) + m(2,2) + 1.0f) / 4.0f;
  if (qx < 0.0f) qx = 0.0f;
  if (qy < 0.0f) qy = 0.0f;
  if (qz < 0.0f) qz = 0.0f;
  if (qw < 0.0f) qw = 0.0f;
  qx = sqrt(qx);
  qy = sqrt(qy);
  qz = sqrt(qz);
  qw = sqrt(qw);
  if (qx >= qy && qx >= qz && qx >= qw)
  {
      qx *= +1.0f;
      q1 *= SIGN(m(2,1) - m(1,2));
      q2 *= SIGN(m(0,2) - m(2,0));
      q3 *= SIGN(m(1,0) - m(0,1));
  }
  else if (qy >= qx && qy >= qz && qy >= qw)
  {
      qx *= SIGN(m(2,1) - m(1,2);
      qy *= 1.0f;
      qz *= SIGN(m(1,0) + m(0,1));
      qw *= SIGN(m(0,2) + m(2,0));
  }
  else if (qz >= qx && qz >= qy && qz >= qw)
```

```
    {
        qx *= SIGN(m(0,2) - m(2,0));
        qy *= SIGN(m(1,0) + m(0,1));
        qz *= 1.0f;
        qw *= SIGN(m(2,1) + m(1,2));
    }
    else if (qw >= qx && qw >= qy && qw >= qz)
    {
        qx *= SIGN(m(1,0) - m(0,1));
        qy *= SIGN(m(2,0) + m(0,2));
        qz *= SIGN(m(2,1) + m(1,2));
        qw *= 1.0f;
    }
    else
    {
        Debug_c.Assert("**error**\n");
    }
    r = NORM(qx, qy, qz, qw);
    qx /= r;
    qy /= r;
    qz /= r;
    qw /= r;
    return Quaternion(qx,qy,qz,qw);
}
```

3.4.4 *Quaternion-Quaternion Multiplication*

You multiply quaternions together to concatenate the rotational transforms (i.e., analogous to how you multiply matrices together to combine the individual transforms into a single unified solution). The quaternion multiplication mathematics is easier to digest, if you subdivide the quaternion elements into a 'scalar' s and 'vector' v component and use the dot and cross product:

$$\begin{aligned}(sa,\vec{va})(sb,\vec{vb}) &= (sa)(sb) + (sa)(\vec{vb}) + (sb)(\vec{va}) + ((\vec{va})\times(\vec{vb})) - ((\vec{va})\cdot(\vec{vb})) \\ &\text{group into parts} \\ &= ((sa)(sb) - ((\vec{va})\cdot(\vec{vb}))), \ \text{scalar part} \\ &\quad ((sa)(sb) + (sb)(\vec{va}) + ((\vec{va})\times(\vec{vb}))) \ \text{vector part}\end{aligned} \tag{3.14}$$

Listing 3.20: Quaternion Quaternion Multiplication.

```

Quaternion Multiplication(const Quaternion& qa,
              const Quaternion& qb)
{
  Quaternion qr = Quaternion.Identity;
  Vector3 va = Vector3(qa.x, qa.y, qa.z);
  Vector3 vb = Vector3(qb.x, qb.y, qb.z);
  qr.w = qa.w*qb.w - Vector3.Dot(va,vb);
  Vector3 vr = Vector3.Cross(va,vb) + qa.w*vb + qb.w*va;
  qr.x = vr.x;
```

```
	qr.y = vr.y;
	qr.z = vr.z;
	return qr;
}
```

3.4.5 *Quaternion Inverse (Conjugate)*

For a unit-quaternion the conjugate is the same as the inverse. You represent the conjugate by the $*$ symbol, e.g., conjugate(q) = q^*.

The conjugate is useful because it has the following properties:

- $q_a^* \; q_b^* = (q_b \; q_a)^*$ In this way you can change the order of the multipicands.
- qq^* = a2 + b2 + c2 + d2 = real number. Multiplying a quaternion by its conjugate gives a real number. This makes the conjugate useful for finding the multiplicative inverse. For instance, if you are using a quaternion q to represent a rotation then conj(q) represents the same rotation in the reverse direction.
- $P_{out} = q \; P_{in} \; q^*$ you use this to calculate a rotation transform.

Listing 3.21: Quaternion Conjugate.

```

Quaternion Conjugate(const Quaternion& q)
{
	// Note, you invert the vector component
	Quaternion qr (q.w, -q.x,-q.y,-w.z);
	return qr;
}
```

3.4.6 *Transform a Vector by a Quaternion*

As pointed out, you can use the Conjugate to make transforming a Vector3 a piece of cake. You convert the Vector3 to a quaternion (i.e., set the scalar W component to 0), then multiply them to get the result. You extract the transformed Vector3 (i.e., the x, y, and z component of the resulting multiplied quaternions). Simple eh?. The formulation is given by:

$$v_{out} = q \; v_{in} \; q^* \tag{3.15}$$

where v_{in} is the original point converted to a quaternion (i.e., w component is set to zero), q and q^* are the quaternion and quaternion conjugate, and v_{out} is the transformed point (i.e., x, y, and z component of the resulting quaternion).

Listing 3.22: Quaternion Vector Transform.

```

Vector3 Transform(const Vector3& v, const Quaternion& q)
{
  Quaternion qv(0, v.x, v.y, v.z);
  Quaternion qr = q * qv * Conjugate(q);
  return new Vector3( qr.x, qr.y, qr.z );
}
```

3.5 *Summary*

Most of the time, you'll use pre-written math libraries (such as, vmath or glm). If you do write a set of math libraries, you'll probably write them once and never need to worry about writing them again. However, having a solid understanding of how the vector mathematics works can dramatically help you understand the creation, optimization, and debugging of algorithms (both from a theoretical and practical perspective).

Type	Representation
Vector $\vec{v}$	x, y, z
Unit Vector $\hat{v}$	$\frac{\vec{v}}{\vert\vert\vec{v}\vert\vert} \implies \vert\vert\hat{v}\vert\vert = 1$
Position $\vec{p}$	x, y, z
Rotation $\vec{q}$	x, y, z, w (quaternion)
Sphere	$\vec{p}$, r
Plane	$\vec{p}$, $\hat{n}$
AABB	$\vec{p}$, $\vec{e}$
OBB	$\vec{p}$, $\vec{q}$, $\vec{e}$
Line/Segment	$\vec{p}_0$, $\vec{p}_1$
Ray	$\vec{p}$, $\hat{n}$
Triangle (t)	$\vec{p}_0$,$\vec{p}_1$,$\vec{p}_2$
Mesh	$\sum t$
Capsule	$\vec{p}_0$, $\vec{p}_1$, r

Table 3.1: Defining primitive objects using a mathematical representation (e.g., a sphere is represented by a centre position p and a radius r).

You use the symbols in Table 3.1, such as, arrows and hats above vectors, to enable us to read mathematical equations at a glance. For instance, you can easily identify a scalar a and a $\vec{a}$ quickly; or a vector $\vec{b}$ and a unit-vector $\hat{b}$. You also provide simple implementation listings to solidify the your understanding.

3.6 *Exercises*

After you're familiar with the core mathematical principles, you'll need to constantly practice to strengthen your understanding. The following example questions provide you this opportunity.

3.6.1 *Chapter Questions*

Question Given the three matrices A: translation along the vector v = (4, 0, 2), B: rotation 90 degrees around the z-axis and C: a non-uniform scaling with 2 in x, 3 in y and 4 in z.

a) Give the (4 x 4) matrix form of each of A, B and C.

b) Calculate the transformed point P', given the point P = (1, 2, 3, 1). i.e., P' = CABP

Question What does it mean if two vectors are orthogonal? How can you determine if two vectors are orthogonal?

Question Give a 3x3 homogeneous matrix to rotate an image clockwise by 90 degrees. Then shift the image to the right by 10 units. Finally scale the image by twice as large. All these transformations are to be done one after the other in sequence

Question What are the basic 2D geometric transformations? Explain each with its matrix representation

Question Show that the composition of two rotations is additive by concatenating the matrix representations for $R(\theta_1)$ and $R(\theta_2)$ to obtain $R(\theta)$. $R(\theta) = R(\theta_1 + \theta_2)$

Question Derive the transformation matrix for rotation about any axis

Question Given a triangle A(0,0), B(1,1) and C(6,2). Write down the transformation matrix to magnify the triangle to twice its size keeping C(6,2) fixed.

Question Explain basic 2D transformations? Give the homogeneous matrix representations for each transformation.

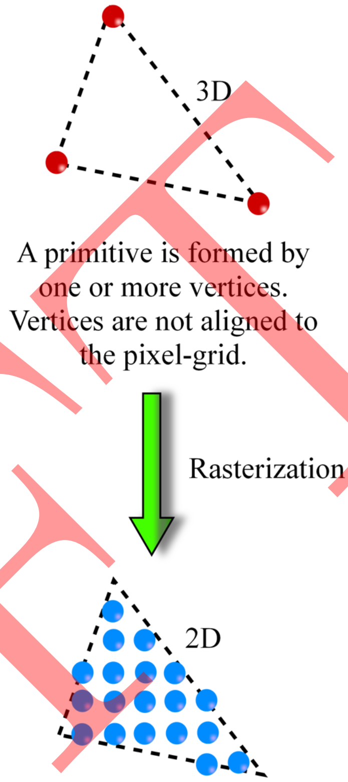

Figure 4.1: How you go from 3-dimensional geometry using transforms and rasterization techniques to produce a graphical output on your screen (pixels).

4
Graphical Principles

As you might be new to graphical programming, you might find a number of concepts confusing and alien when discussed in the context of the Vulkan API, such as, shaders and projection transforms. While it would be beyond the scope of a single Chapter to teach a complete graphical syllabus, instead this Chapter aims to review a number of core graphical principles that are fundamental to most graphical solutions. In addition, you're encouraged to read around the subject to complement your understanding of the material (e.g., computer graphics books, introductory graphics/maths articles and online tutorials). In this Chapter, you'll quickly review the following concepts:

- Basic Types
  Scalars, Vectors, Floats, Colors, ..
- Transforms
  Coordinate Spaces
  Camera and Projection
- Primitives
  Lines, Triangles
- Data/Geometry
  Vertices & Indices
- Drawing Principles
  Draw Ordering (Counter)-Clockwise, Texturing, Depth Buffer, Clipping
  Render output and clipping-cube
- Programmable Graphics
  Shaders, Pipeline, ..

4.1 Basic Types

As with any standard programming language, you'll have a set of standard data types, such as, floats, doubles and strings. You'll also need to create a number of structures to encapsulate data for ease of use and readability. For example, arrays of data for representing your geometry, matrix transforms and color information. You need to be aware of overheads, such as, the sizes of variables in memory, alignment specifics (structure padding) and conversion costs (doubles to floats). For example, see Figure 4.2 for a short list of common Vulkan types and Listing 4.1 for a simple power of two test function.

Vulkan Data Types

Vulkan Data Type	Representation	C-Type
VkFlags	32-bit	uint32_t
VkBool32	32-bit	uint32_t
VkDeviceSize	64-bit	uint64_t
VkSampleMask	32-bit	uint32_t
.....	....	....

Figure 4.2: To help with cross-platform development Vulkan defines various data types that map standard C/C++ data types.

Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 4.3: Matrix representation of basic transformations (2-dimensions) - see Chapter 3 for a review of vector/matrix mathematical concepts.

Listing 4.1: Simple example of understanding binary data - testing if a value is a power of two (e.g., 2, 4, 8, 16, 32, 64, 128 - such as for memory allocations and texture dimensions/widths/heights).

```
bool IsAlignedPowerOfTwo(uint32_t alignment)
{
    // 2- 10
    // 4- 100
    // 8- 1000
    // ...
    // Returns true if a power of 2
  return ((alignment & (alignment-1)) == 0);

    /*
    e.g.
    val          = 1000 (8) - power of 2
    val-1        = 0111
    val & (val-1) = 0000 (==0) return true

    val          = 0110 (6) - not a power of 2
    val-1        = 0101
    val & (val-1) = 0100 (!=0) return false

    */
}// End AlignedPowerOfTwo
```

4.2 *Transforms*

Mathematics has many applications in computer graphics especially matrices as discussed in Chapter 3. Matrices represent groups of equations that provide a compact, efficient and systematic way of doing the mathematical operations, such as, rotation, translation, scaling and projection (i.e., the representation of any transformation affine or non-affine). Importantly, the hardware within the computer (like the GPU) is optimised for matrix arithmetic. Of course, one of the most powerful feature that matrices give you is the ability to concatenate several transformations into a single matrix.

Common vector and matrix graphical operations that you'll come across again and again (and should ideally be comfortable with), include:

- Matrix-Vector Transform
- Matrix-Matrix Multiplication
- Vector Cross/Dot Product
- Rotation Matrix (x, y and z axis)
- Scale Matrix
- Translation Matrix
- Projection Matrix
- View Matrix

For a refresher on basic vector and matrix operations see Chapter 3.

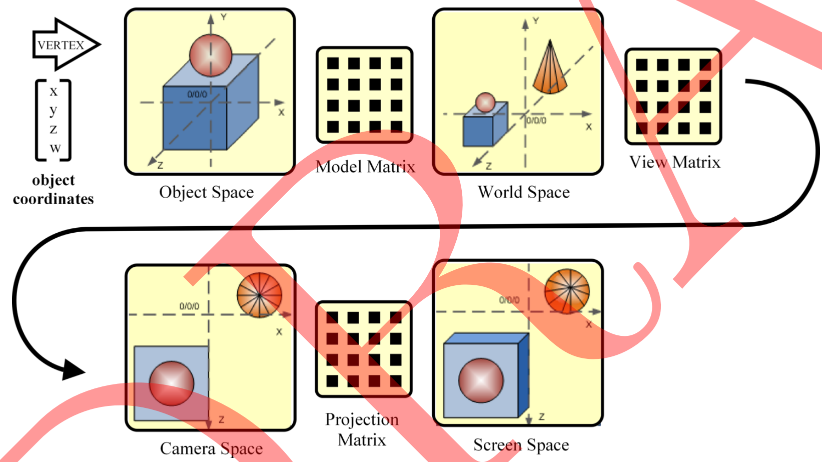

Figure 4.4: Simplified graphical overview of the transformation stages between spaces (local space, world space, camera space and projection space) using matrix transforms (model matrix, view matrix, world matrix and a projection matrix).

As shown in Figure 4.4, there are multiple coordinate systems involved in 3-dimensional graphics, such as, Object Space, World Space (aka Model Space), Camera Space (aka Eye Space or View Space), and Screen Space (aka Clip Space). The best thing is, the conversion between the different transform spaces is effortless. You switch between different spaces by multiplying by a transform matrix. For instance, switching from world space to camera space you'd use your 'view matrix'.

While it's important you know how matrix and vector operations work (especially for 3-dimensional graphics), you don't always have to write your own, and a number of free open source libraries are available. For example, one popular mathematics library is:

OpenGL Mathematics (GLM) [1] library for graphics software based on the OpenGL Shading Language (GLSL) specifications. GLM is a header only C++ mathematics library that provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL.

4.2.1 *Homogeneous Coordinates (or Projective Coordinates)*

Cartesian coordinate transforms, such as, translation and perspective projection, cannot be expressed through matrix multiplication alone and is one of the core reasons you need to use homogeneous coordinate. Your graphics card takes advantage of homogeneous coordinates to perform transforms efficiently using vector processors with 4-element registers (e.g., programmable shaders and pipeline operations) - making matrix operations highly desirable. Any transformation can be represented as a matrix with each matrix having four columns of four rows due to the homogeneous coordinate system (Figure 4.6). In order to position and align your objects and set up representations of your scene inside your computer, you'll need to be able to transform your objects. As pointed out earlier, there are many transformations available to you (like stretching, twisting and bending), but the three absolutely necessary transforms you need to know are rotation, translation and scaling (Figure 4.5).

Identity

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation Around X

$$\begin{bmatrix} \boxed{1} & 0 & 0 & 0 \\ 0 & cos(v) & -sin(v) & 0 \\ 0 & sin(v) & cos(v) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation Around Y

$$\begin{bmatrix} cos(v) & 0 & sin(v) & 0 \\ 0 & \boxed{1} & 0 & 0 \\ -sin(v) & 0 & cos(v) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation Around Z

$$\begin{bmatrix} cos(v) & -sin(v) & 0 & 0 \\ sin(v) & cos(v) & 0 & 0 \\ 0 & 0 & \boxed{1} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 4.5: Core 3-dimensional homogeneous matrix transforms. Matrices are able to represent a variety of geometric transformations - which are able to be combined with each other by matrix multiplication. As a result, any perspective projection of space can be represented as a single matrix.

Remember, when you transforms your points the result is always made homogeneous. This means that your coordinate values are divided with 'W' (see Figure 4.6).

Point transformed by a translation matrix

$$P' = M \cdot P = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & w \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ \boxed{1} \end{bmatrix} = \begin{bmatrix} x+a \\ y+b \\ z+c \\ \boxed{w} \end{bmatrix}$$

and homogeneous

$$\begin{bmatrix} (x+a)/\boxed{w} \\ (y+b)/\boxed{w} \\ (z+c)/\boxed{w} \\ \boxed{1} \end{bmatrix}$$

Figure 4.6: Homogeneous coordinates are crucial in computer graphics and 3D computer vision as they allow affine transformations and, in general, projective transformations to be easily represented by a matrix.

4.2.2 *Normalized Device Coordinates (NDC)*

The Normalize Device Coordinates (NDC) come into action towards the end of the processing (i.e., during the transition from 4D Homogeneous coordinates to screen pixels):

1. Your 4x4 PROJECTION transform takes you from 4D eye coordinates to 4D clip coordinates
2. Then the perspective divide takes you from 4D clip coordinates to 3D NDC coordinates
3. Then the viewport transformation takes those 3D NDC coordinates into 3D window coordinates.

Remember, multiplying by your 'PROJECTION' matrix takes you to 4D CLIP space.

Then the perspective divide gets you to the 3D NDC space.

(EYE-SPACE) –>
(PROJECTION TRANSFORM) –>
(CLIP-SPACE) –>
(PERSPECTIVE DIVIDE) –>
NDC-SPACE

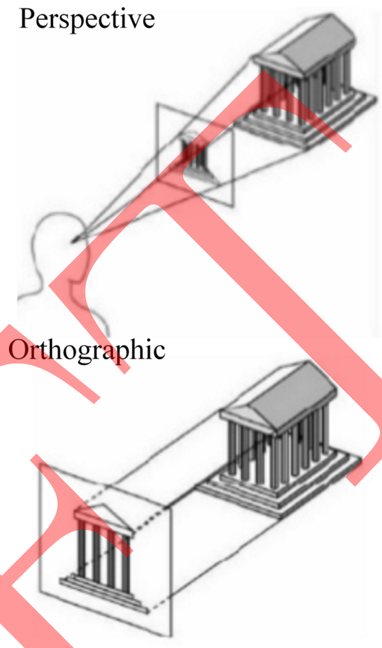

Figure 4.7: 3-dimensional model to a 2-dimensional image.

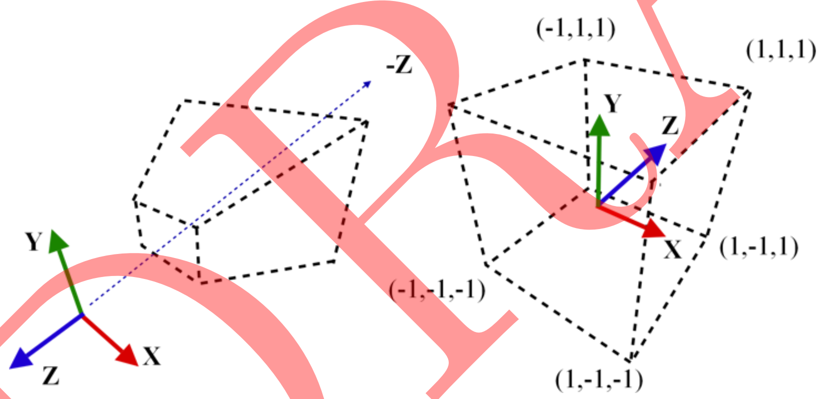

Figure 4.8: Mapping the pyramid frustum (eye coordinates) to the cube Normalize Device Coordinates (NDC).

4.2.3 *Eye Coordinates*

When you transform your geometry by the model and view matrix - this takes you to 'eye coordinates'. In other words, Vulkan defines the camera to be always located at (0, 0, 0) and facing to -Z axis in the eye space coordinates. You transform your vertices (or geometry) from object space to eye space using your 'model-view' matrix which you perform on the GPU in the shader (e.g., vertex shader). The 'model-view' matrix is a combination of the 'Model' and 'View' matrices.

(a) $$\begin{bmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{bmatrix} \mathbf{M_{model}} = \begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ w_{world} \end{bmatrix}$$

(b) $$\begin{bmatrix} x_{world} \\ y_{world} \\ z_{world} \\ w_{world} \end{bmatrix} \mathbf{M_{view}} = \begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{bmatrix}$$

(c) $$\begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{bmatrix} \mathbf{M_{projection}} = \begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix}$$

(d) $$\begin{bmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{bmatrix} = \begin{bmatrix} x_{ncd} \\ y_{ncd} \\ z_{ncd} \end{bmatrix}$$

(e) Viewport (x, y, w, h)
Depth Range (n, f)
$$\begin{bmatrix} \frac{w}{2} x_{ncd} + (x + \frac{w}{2}) \\ \frac{h}{2} x_{ncd} + (y + \frac{h}{2}) \\ \frac{f-n}{2} x_{ncd} + (\frac{f+n}{2}) \end{bmatrix} = \begin{bmatrix} x_{window} \\ y_{window} \\ z_{window} \end{bmatrix}$$

Figure 4.9: Transforms: (a) World Coordinates, (b) Eye Coordinates, (c) Clip Coordinates, (d) Normalized Device Coordinates (NDC), (e) Window Coordinates (Screen Coordinates).

4.2.4 *Projection*

The two main categories of projection are (1) perspective and (2) parallel projection as shown in Figure 4.10. For parallel projections, you'll typically use a basic Orthographic Projection, while for Perspective you'll use something more fancy to capture the real-world perception of objects getting smaller as they get further away. Applications of the the two projection techniques:

- Parallel projection are used for on screen menus or technical drawings
- Perspective projections are used for full 3-dimensional scenes that mimic the real-world (depth and distance)

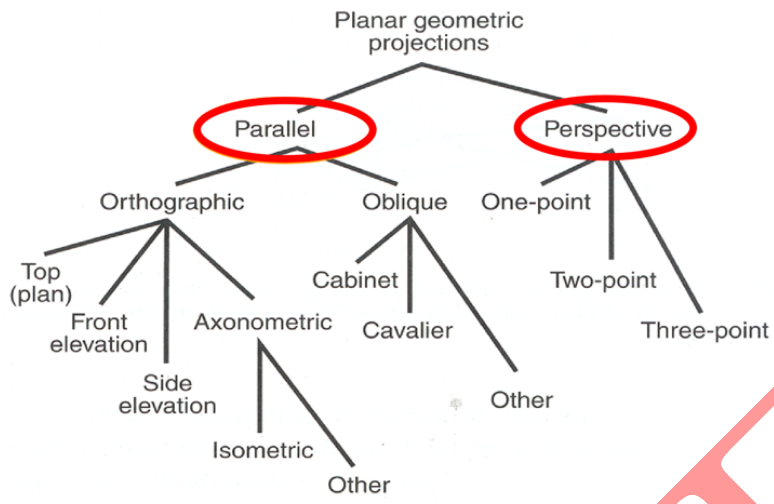

Figure 4.10: Broadly categorising sorting projection methods into two main categories.

4.2.4 *Orthogonal*

A simple orthographic transformation where the original world units would be preserved (the z-coordinate is simply thrown away) is shown below in Equation 4.1:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' = 0 \\ 1 \end{bmatrix} \tag{4.1}$$

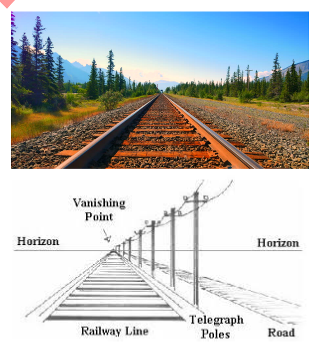

Figure 4.11: Train Track - Linear perspective projection with one vanishing point.

4.2.4 *Perspective*

For perspective transforms, this is closer to what you see in the real-world, where objects closer to viewer look larger and parallel lines appear to converge to single point when they go off into the distance (as with train tracks - Figure 4.11). The mathematics is a little more involved for calculating the projection matrix. However, the principles are governed by simple geometric concepts. As shown in Figure 4.12, the projection matrix works by 'projecting' the object onto a surface from a pin-point camera location. Due to the importance of the projection matrix in computer graphics the steps for calculating a simple projection matrix follow.

You'll typically use a one point perspective - however, multi-point perspective projections are possible (e.g., two and three point). These different linear perspective method use a lines to create the illusion of space on a flat surface. There are three types of linear perspective. One point perspective uses one vanishing point placed on the horizon line. Two point perspective uses two points placed on the horizon line. Three point perspective uses three vanishing points (Figure 4.10).

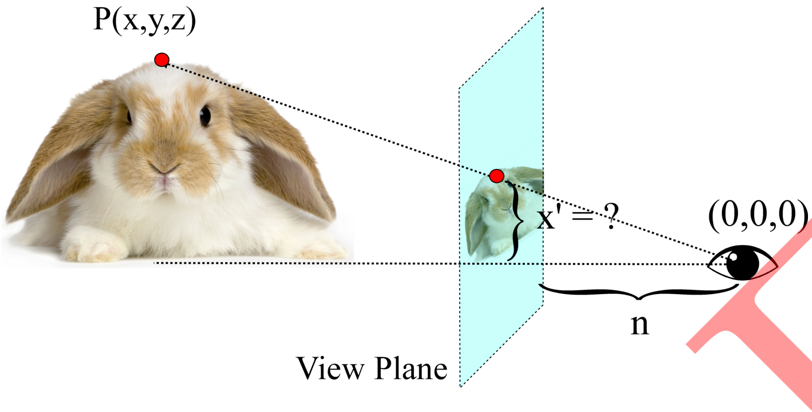

Figure 4.12: Perspective (3D world to 2D screen window). The horizontal and vertical calculations are done independently. The perspective projection calculation uses basic trigonometric principles (e.g., similar triangles) to derive the perspective matrix. For example, in the diagram, you know all of the values except x′ for the projection of the point $P[x, y, z, 1]$ onto the view plane.

As shown in Figure 4.12, the perspective transformation to project the coordinates onto a simple plane is given by Equation 4.2:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/n & 0 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \tag{4.2}$$

where n is the near viewing plane distance (see Figure 4.12). In the perspective case, you use similar triangles to solve for the intersection point on the planes surface.

$$\begin{aligned} \frac{x_c}{n} &= \frac{x_e}{z_e} \\ \frac{y_c}{n} &= \frac{y_e}{z_e} \end{aligned} \tag{4.3}$$

therefore:

$$\begin{aligned} x_c &= \frac{x_e}{z_e/n} \\ y_c &= \frac{y_e}{z_e/n} \end{aligned} \tag{4.4}$$

For a real-world projection matrix, you'd have to specify a number of parameters, and the projection surface may not be square (rectangular with an aspect ratio). You'll now go through the steps to creating a more usable final perspective matrix in detail. You'll start with an empty matrix and add the specifics for each matrix element as you progress through each step.

Step 1 Pass through z_e onto w_c ($w_c = -z_e$):

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} \begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} \tag{4.5}$$

Step 2 Map the input coordinates to the NDC coordinates using the relationship $[l,r]->[-1,1]$, $[b,t]->[-1,1]$:

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ ? & ? & ? & ? \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} \tag{4.6}$$

Step 3 z_c needs to be modified to include depth information for clipping (e.g., depth test) and is 'not' just the near (n) value. Hence, you need to work out how z_e maps to the near-far. Importantly, the z calculation does not depend on the x or y coordinates. Updating the matrix to include to extra variables 'A' and 'B' and solve them:

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} \tag{4.7}$$

$$z_n = \frac{z_c}{w_c} = \frac{Az_e + Bw_e}{-z_e} \tag{4.8}$$

You have one equation and two unknowns, so it's impossible to solve unless you add some additional information. Hence, to accomplish this by specifying the value for z_n when the point is on the near (n) and far (f) planes.

$$\begin{aligned} z_n &= -1 \quad && \text{when} \quad && z_e = -n \\ z_n &= 1 \quad && \text{when} \quad && z_e = f \end{aligned} \tag{4.9}$$

$$\begin{aligned} \frac{-An+B}{n} &= -1 \quad && -> \quad && -An + B = -n \\ \frac{-Af+B}{f} &= 1 \quad && -> \quad && -Af + B = f \end{aligned} \tag{4.10}$$

Hence, you have two equations and two unknowns and should be able to solve for A and B:

$$\begin{aligned} A &= -\frac{f+n}{f-n} \\ B &= -\frac{2fn}{f-n} \end{aligned} \tag{4.11}$$

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} \tag{4.12}$$

Step 4 Simplify the perspective matrix for a general frustum. When the viewing volume is symmetric: $r = -l$ and $t = -b$ it simplifies to:

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} \begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} \tag{4.13}$$

Listing 4.2: Example implementation of a perspective matrix (see Equation 4.13).

```
inline
Matrix4 Perspective(float fov,
                    float aspect,
                    float nearz,
                    float farz)
{
float top = tan(fov * 0.00872664625) * nearz; /* 0.00872664625 = PI/360 */

Matrix4 matrix;
memset(matrix, 0, sizeof(GLfloat) * 16);


matrix[0]  = nearz / (top * aspect);
matrix[5]  = nearz / top;
matrix[10] = -(farz + nearz) / (farz - nearz);
matrix[11] = -1;
matrix[14] = -(2 * farz * nearz) / (farz - nearz);

return matrix;
}// End Perspective(..)
```

4.2.5 *Camera (LookAt)*

In Vulkan, you'll need to explicitly define a camera object for camera transformation. The camera or view matrix is responsible for transforming the entire scene inversely to the origin (0,0,0) and always looking along -Z axis (this space is called eye space).

You construct a view matrix using the LookAt technique. You define the camera location at the eye position, the position you want the camera to look at (or rotating to) the target point target position. You must remember, the eye position and target are defined in 'world space'. The camera LookAt transformation consists of two transformations:

(M_T) translating the whole scene inversely from the eye position to the origin

(M_R) rotating the scene with reverse orientation (MR), so the camera is positioned at the origin and facing to the -Z axis

$$M_{view} = M_R \quad M_T \tag{4.14}$$

The translation part of LookAt transformation is the simplest part to remember as all you need to do is move the camera position to the origin. The translation matrix M_T would be the negation of the eye position.

$$M_T = \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.15}$$

The rotation part of the LookAt transformation requires you to calculate 1st, 2nd and 3rd columns of the rotation matrix.

$$M_R = \begin{bmatrix} l_x & u_x & f_x & 0 \\ l_y & u_y & f_y & 0 \\ l_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} l_x & u_x & f_x & 0 \\ l_y & u_y & f_y & 0 \\ l_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{T} = \begin{bmatrix} l_x & l_y & l_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.16}$$

Finally, the view matrix for camera's LookAt transform is multiplying M_T and M_R together:

$$M_{view} = M_R M_T = \begin{bmatrix} l_x & l_y & l_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} l_x & l_y & l_z & -l_x x_e - l_y y_e - l_z z_e \\ u_x & u_y & u_z & -u_x x_e - u_y y_e - u_z z_e \\ f_x & f_y & f_z & -f_x x_e - f_y y_e - f_z z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.17}$$

Listing 4.3: Unsophisticated LookAt Camera View Implementation.

```
inline
Matrix4 LookAt( const Vector3& eye,
                const Vector3& target,
                const Vector3& upDir)
{
    // calculate the forward vector from target to eye
    Vector3 forward = eye - target;
    forward = Vector3::Normalize(forward); // make unit length

    // calcualte the left vector
    Vector3 left = Vector3::Cross(upDir, forward);   // cross product
    left = Vector3::Normalize(left);

    // recalculate the orthonormal up vector
    Vector3 up = Vector3::Cross(forward, left);    // cross product

    // init 4x4 matrix
    Matrix4 matrix;
    matrix = Matrix4::Identity();

    // set rotation part, inverse rotation matrix: M^-1 = M^T for Euclidean transform
    matrix[0] = left.x;
    matrix[4] = left.y;
    matrix[8] = left.z;
    matrix[1] = up.x;
    matrix[5] = up.y;
    matrix[9] = up.z;
    matrix[2] = forward.x;
    matrix[6] = forward.y;
    matrix[10]= forward.z;

    // set translation part
    matrix[12]= -left.x    * eye.x - left.y    * eye.y - left.z    * eye.z;
    matrix[13]= -up.x      * eye.x - up.y      * eye.y - up.z      * eye.z;
    matrix[14]= -forward.x * eye.x - forward.y * eye.y - forward.z * eye.z;

    return matrix;
};// End LookAt(..)
```

A typical implementation of the LookAt transformation calculation may look something like Listing 4.3.

4.3 *Primitives*

Primitives are the basic drawing elements (the building blocks for more complex geometry). For example, the most common and simplest primitive is the triangle. However, other simple primitives includes, points, lines, and even squares. In Vulkan, you need to specify the primitive type you'll be using, so the render output knows how to interpret your stream of data (e.g., three points for a triangle or two points for a line).

1. Lines

Lists, Strips, Fans, ..

2. Triangles

Lists Strips, Fans, ..

The primitive topology is described in Vulkan via the VkPrimitive-Topology enumerated type as shown below in Listing 4.3:

```
typedef enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST                    = 0,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST                     = 1,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP                    = 2,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST                 = 3,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP                = 4,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN                  = 5,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY      = 6,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY     = 7,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY  = 8,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 9,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST                    = 10,
    VK_PRIMITIVE_TOPOLOGY_BEGIN_RANGE = VK_PRIMITIVE_TOPOLOGY_POINT_LIST,
    VK_PRIMITIVE_TOPOLOGY_END_RANGE   = VK_PRIMITIVE_TOPOLOGY_PATCH_LIST,
    VK_PRIMITIVE_TOPOLOGY_RANGE_SIZE  = (VK_PRIMITIVE_TOPOLOGY_PATCH_LIST -
          VK_PRIMITIVE_TOPOLOGY_POINT_LIST + 1),
    VK_PRIMITIVE_TOPOLOGY_MAX_ENUM                      = 0x7FFFFFFF
} VkPrimitiveTopology;
```

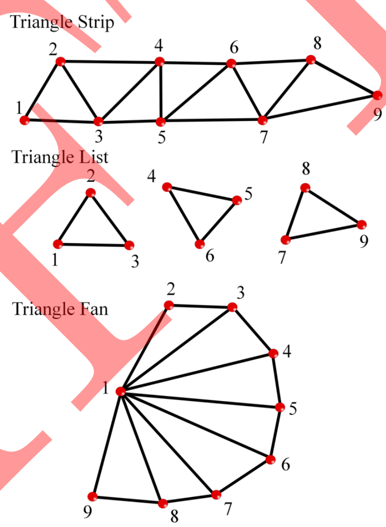

Figure 4.13: Format and configuration of the geometric data needs to be specified.

4.3.1 *Backface Culling (Clockwise/Counter-Clockwise)*

The draw order enables the graphical API to 'cull' unseen triangles (i.e., triangles have two sides - front and back - the triangles facing away from the viewer aren't drawn). Hence, you need to define the preferred drawing order when you initialize Vulkan (or any other graphical API). The draw order is described in Vulkan via the Vk-FrontFace enumerated type as shown below in Listing 4.3.1:

```
typedef enum VkFrontFace {
    VK_FRONT_FACE_COUNTER_CLOCKWISE = 0,
    VK_FRONT_FACE_CLOCKWISE         = 1,
    VK_FRONT_FACE_BEGIN_RANGE       = VK_FRONT_FACE_COUNTER_CLOCKWISE,
    VK_FRONT_FACE_END_RANGE         = VK_FRONT_FACE_CLOCKWISE,
    VK_FRONT_FACE_RANGE_SIZE        = (VK_FRONT_FACE_CLOCKWISE -
          VK_FRONT_FACE_COUNTER_CLOCKWISE + 1),
    VK_FRONT_FACE_MAX_ENUM          = 0x7FFFFFFF
} VkFrontFace;
```

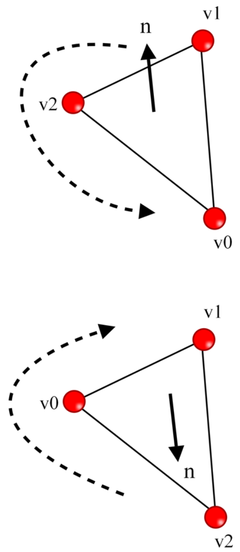

Figure 4.14: Backface culling removes (doesn't draw) triangles that are facing away from the viewer. The direction of the triangle (front/back) is determined by the winding order.

To distinguish between the two sides you use the following convention (see Figure 4.14):

$$\begin{aligned} e0 &= v1 - v0 \\ e1 &= v2 - v0 \\ n &= \frac{e0 \times e1}{||e0 \times e1||} \end{aligned} \tag{4.18}$$

where $v0$, $v1$ and $v2$ are the three corner positions of the triangle, $e0$ and $e1$ are the edges of the triangle and n is the triangle normal. The side the normal vector emanates from is the front side and the other side is the back side. You say the triangle is front-facing if the viewer (camera) sees the front side of the triangle, while the triangle is back-facing if the viewer sees the back side. Importantly, the front or back facing direction is determined by the 'ordering' of the vertices. This is not hard-coded either - as you set the ordering in the Vulkan API (i.e., the way you compute the triangle normal), a triangle ordered clockwise (with respect to that viewer) is front-facing, and a triangle ordered counter-clockwise (with respect to that viewer) is back-facing.

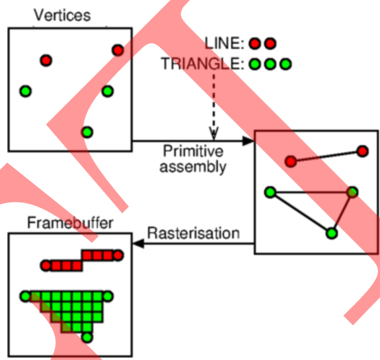

Figure 4.15: Simplified illustration of Vertices to Pixels.

In reality, most 3-dimensional meshes are solid (totally enclosed). The object is constructed so the outside surface of the object has the triangle normals facing outwards. Resulting in the camera seeing the front-facing triangles of a solid object while the back-facing triangles are occluded (culled).

In addition to defining the front facing triangle draw order you also must explicitly define the culling mode. The culling mode is described in Vulkan via the VkCullModeFlagBits enumerated type as shown below in Listing 4.3.1:

```
typedef enum VkCullModeFlagBits {
    VK_CULL_MODE_NONE               = 0,
    VK_CULL_MODE_FRONT_BIT          = 0x00000001,
    VK_CULL_MODE_BACK_BIT           = 0x00000002,
    VK_CULL_MODE_FRONT_AND_BACK     = 0x00000003,
    VK_CULL_MODE_FLAG_BITS_MAX_ENUM = 0x7FFFFFFF
} VkCullModeFlagBits;
```

You'll apply these enumeration types in later Sections when you implement your Vulkan graphical application (e.g., when constructing the Vulkan render pipeline in Listing 6.13).

4.4 Data/Geometry

The basic building block of all 3D object (and scenes) is typically a triangle. A triangle can be created by connecting 3 points or vertices to each other (in 2D or 3D). More complex shapes can be created by adding and assembling more triangles. The geometry can be stored in various file formats or generated procedurally. In addition, the triangles may have color information, texture details and even lighting specific data added to them to generate highly realistic outputs.

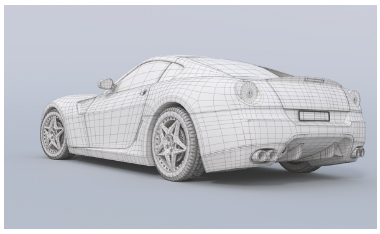

Figure 4.16: Wireframe render of a high-poly car with no materials (i.e., basic lighting) to illustrate the decomposition of a model as simpler primitives/triangles.

In this book, you'll use simple geometry, such as, triangles, planes and cubes to demonstrate various graphical techniques. However, you'll eventually want to draw more complex models/scenes (e.g., Figure 4.16). Of course, manually typing in the vertex/color information for details meshes would be insane. In computer graphics there are usually lots of complicated and interesting models freely available which are prettier to look at than simple planes and cubes.

While you might want to write your own model loading implementation, a quick solution is to take advantage of popular free open source solutions solution. For example, one such model loading library is:

Open Asset Import Library (short name: Assimp) [2] which is a portable Open Source library to import various well-known 3D model formats in an uniform manner. Assimp is able to import dozens of different model file formats by loading all the model's data into generalized data structures. As soon as Assimp has loaded the model, you can retrieve all the data you need from the data structures and convert them to your Vulkan specified layout. This becomes valuable once you've got your Vulkan application up and running and you want to start adding to its functionality.

4.5 *Drawing Principles*

In Vulkan (and with other modern graphical API), the viewing frustum is mapped to a cube that extends from -1 to 1 in the x, y and z (see Figure 4.17. Note, you can flip the z-axis to create a left handed coordinate system during projection transformation discussed in previous Sections when you convert from 3D to 2D.

1. Data (triangles) are passed to the renderer
2. Transforms are applied (on the shader) to the vertices (triangles)
3. The 'rasterization' process draws the geometry to the image (back-buffer screen)
4. Various optimisations/enhancements take place:
   Depth buffer so geometry is drawn in the correct order
   Culling so only clockwise (or counter-clockwise depending upon the configuration) triangles are drawn (i.e., backface culling)
   Clipping (the output render frustum is mapped to a -1 to 1 clip space region)

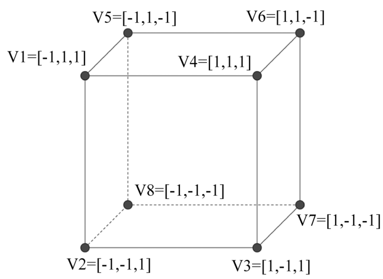

Figure 4.17: Pass-through (identity-transform) for the graphics renderer will output primitives within the **clip space clip cube**.

4.6 *Programmable Graphics & Shaders*

Shaders basically give you the ability to customize your graphics card (akin to programming your CPU). The GPU has different programmable stages that are specifically optimised to perform specific operations (e.g., vertex level or pixel level). For instance, the vertex shader transforms all the vertices positions in virtual space (your 3D model space) to the 2D coordinate which appear on screen (2D screen space). The fragment shader basically gives you the ability to manipulate the pixel information, such as, the pixel color/brightness.

In addition to the vertex and pixel shader there are other shader types. The shader types available in Vulkan are accessed via the VkShaderStageFlagBits enumerated type as shown below in Listing 4.6:

```
typedef enum VkShaderStageFlagBits {
    VK_SHADER_STAGE_VERTEX_BIT                  = 0x00000001,
    VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT    = 0x00000002,
    VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT = 0x00000004,
    VK_SHADER_STAGE_GEOMETRY_BIT                = 0x00000008,
    VK_SHADER_STAGE_FRAGMENT_BIT                = 0x00000010,
    VK_SHADER_STAGE_COMPUTE_BIT                 = 0x00000020,
    VK_SHADER_STAGE_ALL_GRAPHICS                = 0x0000001F,
    VK_SHADER_STAGE_ALL                         = 0x7FFFFFFF,
    VK_SHADER_STAGE_FLAG_BITS_MAX_ENUM          = 0x7FFFFFFF
} VkShaderStageFlagBits;
```

Multiple shader flags, such as, 'VK_SHADER_STAGE_ALL_GRAPHICS' will become apparent when you start programming the graphical effects with the Vulkan API in later Chapters.

As you might be starting to see, most stages feed their output directly onto the next stage of the pipeline (hence the name 'pipeline). For instance, the Vertex Shader stage inputs data from the Input Assembler stage, does its own work, and then outputs its results to the Geometry Shader stage (see Figure 4.18).

- **Input Assembler Stage** The start of the pipeline - reads geometric data (vertices and indices) from memory and uses it to assemble geometric primitives (such as, triangles and lines)
- **Vertex Stage** After the primitives have been assembled, the vertices are fed into the vertex shader stage. The vertex shader can be thought of as a function that inputs a vertex and outputs a vertex (one vertex in and one vertex out).
- **Tessellator Stage** As the name indicates, this stage is responsible for tessellation - that is this stage subdivides the triangles of a mesh to add new triangles. These new triangles can then be offset into new positions to create finer mesh detail
- **Geometry Stage** The geometry shader stage is optional. You'll learn about the geometry shader in Chapter 9. The geometry shader inputs entire primitives. For example, if you were drawing triangle lists, then the input to the geometry shader would be the three vertices defining the triangle. Crucially, the geometry shader is able to create and destroy geometry (unlike the vertex stage). For example,

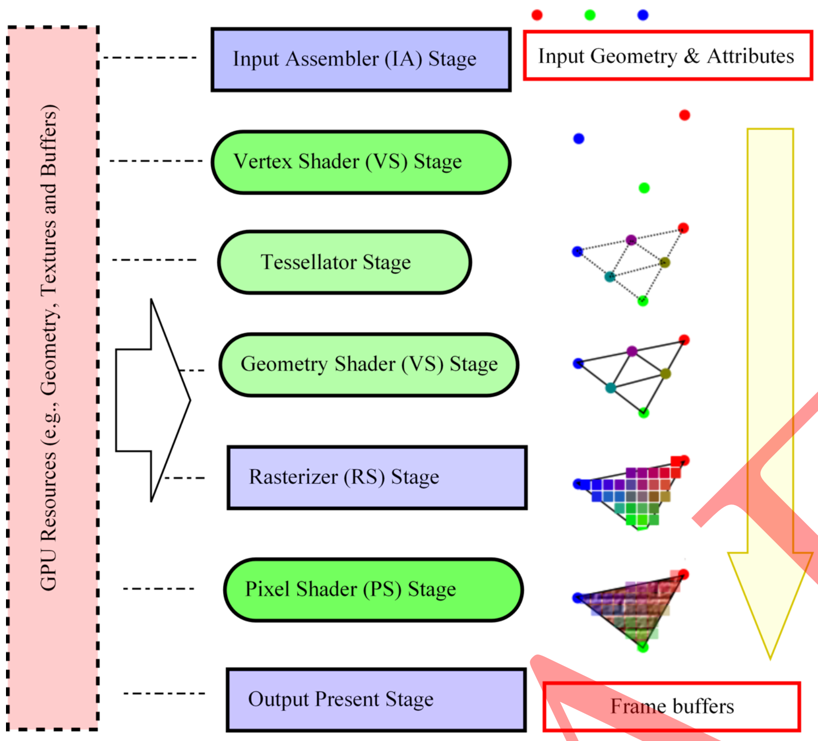

Figure 4.18: Common rendering pipeline stages. The arrow going from the GPU resources to the different stage means different stages can access the resources as input. For instance, the pixel shader stage may need to read data from a texture resource stored in memory in order to do its work. Also the downward arrow indicates the 'pipeline' flow from stage to stage.

the input primitive can be expanded into one or more other primitives, or the geometry shader can choose not to output a primitive based on some condition. You'd be able to pass in a single vertex to the geometry shader and output an entire geometric shape (or no shape at all)

- **Rasterization Stage** The main job of the rasterization stage is to compute pixel colors from the projected 3D triangles
- **Pixel (or Fragment) Stage** A pixel or fragment shader is executed for each pixel fragment and uses the interpolated vertex attributes as input to compute a color. A pixel shader can be as simple as returning a constant color, to doing more complicated things like per-pixel lighting, reflections and shadowing effects
- **Final Output Stage** After pixel fragments have been generated by the pixel (fragment) shader, they move onto the final output stage of the rendering pipeline. In this stage, some pixel fragments may be rejected (e.g., from the depth or stencil buffer tests). Pixel fragments that are not rejected are written to the back buffer. Blending is also done in this stage, where a pixel may be blended with the pixel currently on the back buffer instead of overriding it completely

4.7 Exercises

A number of well written books are available on the principles of computer graphics which complement this text (e.g., mathematics and geometry). Once you've got your Vulkan graphical application up and running you'll be able to extend your simple implementation developed in this book to encapsulate advanced features, such as, ambient occlusion, instancing, tessellation shader and post-processing.

Recommended books specifically focusing on Graphical Principles and Mathematics include:

- 3D math primer for graphics and game development by Dunn, Fletcher and Parberry, Ian [5]
- Computer Graphics: Principles and Practice (3rd Edition) by John Hughes et al. [7]
- Real-Time Rendering by Tomas Akenine-Moller et al. [3]
- 3D Graphics Programming: Games and Beyond by Sergei Savchenko [10]

4.7.1 Chapter Questions

Question What is the clip space?

Question Why are matrices used in graphical transforms?

Question What is a primitive?

Question What is the depth buffer used for?

Question Mention three differences between real-time graphics and off-line (photo-realistic) computer graphics. In this context, also explain why graphics hardware, e.g., graphics cards are useful for computer graphics

Question In the context of the graphics pipeline, describe the responsibility of the vertex shader, rasterizer and pixel shader stage of the graphics pipeline.

Question Mention three coordinate systems (spaces) that you may encounter in a rendering pipeline. Briefly explain the purpose of each system.

Question What is backface culling, why is it useful and where in the graphics pipeline can a backface culling test be executed?

Question What is a viewing frustum?

Question Why is the triangle strip more desirable geometric primitive than a list of triangles?

Question What is the difference between convex and concave objects?

Question For the eye position e=[0,2,0] and a target position t=[0,-1,0] and a view-up vector up=[1,1,0], what is the camera transformation matrix?

Question Write the perspective projection matrix. Multiply it by the given homogeneous point to demonstrate how it generates pixel coordinates that reflect perspective foreshortening:

$$\begin{bmatrix} \quad , & \quad , & \quad , & \quad \\ \quad , & \quad , & \quad , & \quad \\ \quad , & \quad , & \quad , & \quad \\ \quad , & \quad , & \quad , & \quad \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{matrix} \text{Transformed 4D point} \\ [\quad , \quad , \quad , \quad] \end{matrix} \qquad \begin{matrix} \text{Pixel value} \\ \text{becomes} \end{matrix} \qquad [\qquad , \qquad]$$

Question Explain the differences between 'raster' and 'vector graphics.

Question Distinguish between window port and viewport.

Question A cube is placed at the origin of a 3D system. Such that all its vertices have positive coordinate values and sides are parallel to the three principal axes. Indicate a convenient position of a viewer at which he can see a 2-point perspective projection.

Question Define vanishing points. Is the location of the vanishing point directly related to the viewing point?

Question What are the various logical graphic input primitives? What are the various input modes?

Question What are the different projection methods? Explain

Question Explain RGB and HSV color modelling?

Question What is homogeneous co-ordinate? Why is a homogeneous co-ordinate system needed in transformation matrix?

Question Derive the transformation matrix for perspective projection.

Question Explain the transformations with examples: (i) Reflection. (ii) Shear.

Question Explain what the Depth-Buffer method is and why we need it?

Question Explain parallel and perspective projections.

Question Discuss the Back-Face surface removal algorithm.

Question Explain depth buffer for visible surface detection in 3D graphics.

Question What is view volume? How is it specified?

Question Discuss the Back-Face surface removal algorithm.

Question Explain window to viewport transformation.

5

Shaders

```
vec3 gamma = vec3(1.0/2.2, 1.0/2.2, 1.0/2.2);
finalColor = vec3(pow(linearColor.r, gamma.r),
                  pow(linearColor.g, gamma.g),
                  pow(linearColor.b, gamma.b));

vec3 gamma = vec3(1.0/2.2);
finalColor = pow(linearColor, gamma);
```

5.1 Introduction

Shaders are the chocolate sauce on your ice-cream. They offer truly limitless possibilities. Shader are in almost all recent real-time graphical applications (like video games), not to mention, animated CGI movies. Some popular techniques that use shaders are: parallax-mapping (bump-mapping), phong-shading, cell-shading, bloom and high dynamic range lighting (HDR). So what are shaders? Shaders are small programs developed by 'you' with the ability to customize the graphical pipeline, such as:

- transform data (manipulate your geometry)
- determine colors (principles of light)
- animate and move data
- and much more

Once upon a time, long ago, graphical processing units had static pipelines that were configurable through flags and states using the configurable API. This, of course, stunted the creative juices of developers (prevented any customization). As time progressed OpenGL and DirectX (and now Vulkan) solved this problem by making the pipeline 'programmable' (initially via low-level assembly shader languages and later high level languages like GLSL (OpenGL Shading Language) and HLSL (High-Level Shader Language)).

Currently there are three major shader languages:

- Cg (Nvidia)
- HLSL (Microsoft)
  Derived from Cg
- GLSL (OpenGL)

Note you're still able to write shaders in assembly for highly optimised solutions but it's less common due advancements in compiler technologies and computational processing power. The main influences on the development and steering of these shader languages over the years have been the C-language and pre-existing solutions developed in universities and industry with the HLSL coming from Microsoft in 2002 and later GLSL for OpenGL ARB in 2003.

Example applications of vertex shaders (run on per-vertex level) include:

- Color
- Texture
- Position
- Do not change the data type (pass-through)

Example applications of fragment shaders (pixel shaders - run on per-pixel level) include:

- Lighting values
- Output certain color
- Computationally expensive for complex effects due to per-pixel calculation (i.e., every pixel vs ever vertex in the vertex shader)

Example applications of the geometry shaders (manipulates graphical primitives to create new primitives - points, lines and triangles) include:

- Shadow volumes
- Cube map (skybox)

The aim of this chapter is to provide concept and language fundamentals essential to high level shader languages common for graphical processing (e.g., GLSL 4+) (i.e., not provide a full in depth programming guide on shader programming). For those of you who are new to shader programming, there are several resources available, such as, books and web coding sandboxes (for instance Shader Toys, GLSL Sandbox and Vertex Shader Art), that are recommended for learning and developing your shader programming skills. Furthermore, shader experience gained through using different programming interfaces, platforms and tool-kits can be easily translated between the different APIs/tools (Vulkan, DirectX and OpenGL) - large amount of overlap and similarity.

5.1.1 *Anatomy of Shaders*

Shader is a program written in textual form (human readable). You'll find these small (shader) programs have these parts:

- Global variables
- Functions

  Local variables (also variables passed through functions)
- Means of pass arbitrary data from Application to Shader (e.g., uniforms)
- Data structure definitions

Current shaders are written in C-type languages. Vulkan only accepts the Spir-V shader format, however, GLSL shader files can be compile to Spir-V files using the shader SDK compiler (e.g., glslangValidator.exe). Hence, this chapter will focus on GLSL examples, which can be compiled and implemented with the Vulkan API, and should be straightforward to port to other API (DirectX HLSL or existing OpenGL implementations). The compiled Spir-V files will typically have the extension '.spv'. The options for the glslangValidator compiler are given below in Listing 5.1:

Listing 5.1: glslangValidator command prompt options.

```
Usage: glslangValidator [option]... [file]...

Where: each 'file' ends in .<stage>, where <stage> is one of
    .conf   to provide an optional config file that replaces the default configuration
            (see -c option below for generating a template)
    .vert   for a vertex shader
    .tesc   for a tessellation control shader
    .tese   for a tessellation evaluation shader
    .geom   for a geometry shader
    .frag   for a fragment shader
    .comp   for a compute shader

Compilation warnings and errors will be printed to stdout.

To get other information, use one of the following options:
Each option must be specified separately.
  -V          create SPIR-V binary, under Vulkan semantics; turns on -l;
              default file name is <stage>.spv (-o overrides this)
              (unless -o is specified, which overrides the default file name)
  -G          create SPIR-V binary, under OpenGL semantics; turns on -l;
              default file name is <stage>.spv (-o overrides this)
  -H          print human readable form of SPIR-V; turns on -V
  -E          print pre-processed GLSL; cannot be used with -l;
              errors will appear on stderr.
  -c          configuration dump;
              creates the default configuration file (redirect to a .conf file)
  -d          default to desktop (#version 110) when there is no shader #version
              (default is ES version 100)
  -h          print this usage message
  -i          intermediate tree (glslang AST) is printed out
  -l          link all input files together to form a single module
```

```
-m          memory leak mode
-o <file>   save binary into <file>, requires a binary option (e.g., -V)
-q          dump reflection query database
-r          relaxed semantic error-checking mode
-s          silent mode
-t          multi-threaded mode
-v          print version strings
-w          suppress warnings (except as required by #extension : warn)
```

In order to understand how shaders work and how you'll use them to extend the drawing capabilities of graphical processing, it is necessary to have an overview of the key concepts of shader programming, first in general and then from the point of view of the graphical processing unit (GPU).

Initially, there was only two programmable stages (e.g., vertex and fragment stages) but as the thirst for freedom continued to grow - more and more stages and control has been given to developers. In this book, the four main shader stages you'll explore are:

- Vertex Stage (Per-Vertex Processing) - e.g., transforming geometry (vertices) to their final space/position
- Fragment Stage (Per-Fragment Processing) - e.g., providing coloring information to the pixel
- (Optional) Geometry Stage - e.g., extends the Vertex Stage with the added ability to add/remove geometry (also able to know about neighbouring primitives)
- (Optional) Tessellation Stage - e.g., ability to add detail to the geometry (add/remove triangles)

These different stages are not static but programmable. These stages are controlled by programs known as 'shaders'. Importantly, you have to implement the 'vertex' and 'fragment' shader (compulsory shaders required to output to the screen), while the geometry and tessellation stages are optional extras (e.g., you don't need to implement them to generate graphical renders). The shaders responsible for processing the different stages are all compiled using the same 'glslangValidator.exe' (see Listing 5.1).

The extension for the shader programs are:

.vert Vertex Shader
.frag Fragment Shader (or Pixel Shader)
.geom Geometry Shader (Chapter 9)
.tesc Tessellation Shader - Control Stage (Chapter 14)
.tese Tessellation Shader - Evaluation Stage (Chapter 14)

Even though the graphical pipeline has changed from a static to a programmable paradigm each stage of the pipeline is still responsible for their original tasks (e.g., transforms and lighting).

5.2 *Link between Vulkan and Shaders*

The Vulkan API has two fronts. The client-side and the server-side. The Vulkan API operates on your application side (client-side), while the shaders operate on the GPU side (server-side). One important responsibility of Vulkan is to link the data to the shaders' (e.g., using layouts and uniforms).

Data in your application is transported to the GPU. Once on the GPU this data your shader will be able to use this data. Your data is linked to your shader through attributes (e.g., specify bindings and locations).

Listing 5.2: Vertex Shader.

```
// shader version
#version 420

// 1. input attribute from your program declared as 'inPos'
layout( location = 0 ) in vec4 inPos;

// 2. Uniforms for the model-view-projection transforms
layout ( binding = 0 ) uniform buffer
{
  mat4 inProjectionMatrix;
  mat4 inViewMatrix;
  mat4 inModelMatrix;
};

// 3. shader program entry point
void main()
{
  // combine the matrices
  mat4 mvp = inModelMatrix * inViewMatrix * inProjectionMatrix;

  // transform position by matrices
  gl_Position = inPos.xyz * mvp;
}// End main(..)
```

The shader version number at the top of each shader file (e.g., #version 420) allows you to know what features/syntax are used by your shader. When no shader version is specified, the default is ES version 100 (#version 110).

Shaders files follow the standard C/C++ commenting syntax (allowing you to make notes/explain your shader code):

- /* comment */ - starting and ending markers for comments (suitable for multiple line comments)
- // - everything after the double slash until a newline is a comment (suitable for single line comments)

5.3 *Linking data to Uniforms*

Linking data to Uniforms is very similar to linking data to attributes. Uniform variables are those that remain constant for each vertex in the scene. You create a buffer (allocate memory on the GPU for the uniforms). You then specify the location of the uniform in the shader. Once you know the location, you provide data to the uniform (e.g., lock and copy the data across). The model, view and projection matrices for transformations fall in this category, since each vertex in the scene is affected by the same model/view/projection matrices.

```
// ** 1 ** Creation of buffer/uniform

// Create 'uniform' buffer for passing constant
// data to the shader (connecting shader with the data)

// create our uniforms buffers:
VkBufferCreateInfo bufferCreateInfo;
memset(&bufferCreateInfo, 0, sizeof(bufferCreateInfo));
bufferCreateInfo.sType          = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
// size in bytes
bufferCreateInfo.size           = sizeof(stBuffer);
bufferCreateInfo.usage          = VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT;
bufferCreateInfo.sharingMode    = VK_SHARING_MODE_EXCLUSIVE;

VkResult result =
vkCreateBuffer( device,
                &bufferCreateInfo,
                NULL,
                outBuffer );
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create uniforms buffer." );

// ** 2 ** Allocate memory for buffer:
VkMemoryRequirements bufferMemoryRequirements = {};
vkGetBufferMemoryRequirements( device,
                               *outBuffer,
                               &bufferMemoryRequirements );

VkMemoryAllocateInfo matrixAllocateInfo  = {};
matrixAllocateInfo.sType                 = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
matrixAllocateInfo.allocationSize        = bufferMemoryRequirements.size;

VkPhysicalDeviceMemoryProperties memoryProperties;
vkGetPhysicalDeviceMemoryProperties( physicalDevice, &memoryProperties );

for ( uint32_t i = 0; i <VK_MAX_MEMORY_TYPES; ++i )
{
    VkMemoryType memoryType = memoryProperties.memoryTypes[i];
    // is this the memory type we are looking for?
    if ( ( memoryType.propertyFlags & VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ) )
    {
      // save location
      matrixAllocateInfo.memoryTypeIndex = i;
      break;
    }
}// End for i
```

```

result =
vkAllocateMemory( device, &matrixAllocateInfo, NULL, outMemory );
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to allocate uniforms buffer memory." );

result =
vkBindBufferMemory( device, *outBuffer, *outMemory, 0 );
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to bind uniforms buffer memory." );

// ** 3 ** Lock/update the uniforms contentsline
uint8_t *pData;
DBG_ASSERT_VULKAN(vkMapMemory(device, memory, 0, sizeof(ubo), 0, (void **)&pData));
memcpy(pData, &ubo, sizeof(ubo));
vkUnmapMemory(device, memory);
```

5.3.1 *Qualifiers*

Variables in shaders take on different behaviours. Some variables can only receive data, others can only provide data. In fact, some of these variables can only be used in the Vertex Shaders, while other variables can only be used in the Geometry or Fragment Shaders. To differentiate these type of variables there are Qualifier Types. For example:

- in/out
- uniforms
- varying

The in/out shader qualifier defines the receiving/sending of data from buffers and whose value may change frequently.

5.3.2 *Uniforms*

A Uniform is a shader qualifier whose value may rarely change. You can think of uniforms as global variables which can be seen by all shader types.

5.3.3 *Varying*

There are times when attribute data needs to be used in different stages of the pipeline (different shaders). In this cases, special type of qualifiers called Varying are used. They take attribute data from the current shader and pass them along to the next shader stage.

5.4 *Developing Shaders*

You're now in the position to develop your own shaders. You're are going to write source code for a vertex and fragment shader. The shader implementations are written in a C-style format. Hence, you can use your favourite text editing program (e.g., notepad or Visual Studio).

The Listing 5.3 below shows the simple.vert file contents. This file will contains your vertex shader source code. Your vertex shader will simply receive vertex data through the input. You'll also receive a Model-View-Projection matrix through a uniform (MVP). You'll then transform the vertex positions by this matrix and set it as the output of the shader. As you'll notice in the vertex shader below, you provide the result to the output of the shader 'gl_Position'. This is a built in variable for the graphical pipeline (i.e., non-programmable aspects of the pipeline - for instance, determining clipping/calculating specific data for the next stage).

Listing 5.3: Basic Vertex Shader.

```
// vertex shader version
#version 420

// input vertex data (i.e., single position)
layout (location = 0) in  vec4 inPos;

// single uniform parameter (transforms) - shared by all vertices
layout (binding = 0) uniform UBO
{
  mat4 MVP;
} ubo;

void main()
{
    // transformed vertex position for the next stage
  gl_Position = ubo.MVP * inPos;
}// End main(..)
```

Next the file called simple.frag shown below Listing 5.4 is the fragment shader:

Listing 5.4: Basic Fragment Shader.

```
// fragment version
#version 420

// output for this fragment (single pixel color)
layout (location = 0) out vec4 outFragColor;

void main()
{
    // constant color - white - all the triangles/primitives would be white
  outFragColor = vec4(1);
```

```
}// End main(..)
```

The shaders above need to compile prior to be loading and using by the Vulkan API (i.e., binary file that is readable by the GPU). When compiling your shader files, ensure you check the output for errors (e.g., typing errors, like spelling mistakes or missing semi-colons). If there are errors in your shader text file your shader compiler will not generate a binary (check the compiler output to ensue it says 'successful').

Common data types:

- int, float, bool, void
- vec2, vec3, vec4
- ivec2, ivec3, ivec4
- mat3, mat4
- sampler2D

For vectors you use the following accessors: 'xyzw' or 'rgba' (including combinations, such as, .xy, .xyz) - this makes the shader implementation very flexible and compact.

Examples:

```
// mat4 to mat3
mat3 viewMatrix = mat3(inViewMatrix);

// selecting a row from a matrix and convert it to a vector
vec3 eye = -inViewMatrix[3].xyz;

// combine the matrices (multiply matrices together)
mat4 mvp = inModelMatrix * inViewMatrix * inProjectionMatrix;

// converting between types (explicitly)
gl_Position = vec4(inPos.xyz, 1.0) * mvp;
```

Common built in functions:

- max
- min
- clamp
- mix
- normalize
- length
- dot
- cross
- texture
- reflect
- pow
- transpose

- inverse
- cos
- sin
- tan
- sqrt

Subsequent chapters you'll implement different shader techniques that enable you to understand the concepts in greater detail, such as, texturing, lighting and geometrical manipulation.

You can create your own structures (i.e., using the 'struct' definition) and your own functions to make your code more manageable and scalable (i.e., you don't need to repeat code but create reusable functions - readability).

5.5 *Summary*

At the end of this chapter you should be starting to see the incredible power of shaders. With little information shaders are able to create an infinite number of possibilities. In following chapters, you'll be delving much deeper into what you can do with shaders. Everything drawn on the screen has been processed by the appropriate 'shader' running behind the scenes. Modifying shaders incorporates a new set of functions and variables allows you to replace the default techniques with your own. This opens up many exciting possibilities: rendering 3D scenes using more creative and sophisticated solutions and algorithms.

5.6 *Exercises*

After you're familiar with the shaders, you'll need to constantly practice to strengthen your understanding. The following example questions provide you this opportunity.

5.6.1 *Chapter Questions*

Question What is the advantage of a programmable pipeline vs a fixed pipeline?

Question In addition to the 'vertex shader' and the 'fragment shader' - name two additional pipeline shaders?

Question What are 'uniforms' for and how and when would you use a uniform?

Question Write a very basic 'vertex' and 'fragment' shader (which transforms the vertex coordinates and outputs colored geometry).

Question In your shader how would you convert a 'vec3' to a 'vec4'?

Question In your shader how would you convert a 'mat4' to a 'mat4'?

Question In your shader how would you extract row from a mat4 (i.e., vec4)?

6

Programming (11 Steps)

Figure 6.1: Example listings use the Microsoft Windows API for the platform specific details.

Welcome to the first coding steps to writing your Vulkan application. This section, you'll learn how to put together the various API elements in context. Essentially, you'll take a difficult and somewhat overwhelming task and develop a set of clear easy to understand functions. The implementation in this book has been broken down into 11 easy steps - making the implementation more manageable. Writing a native Vulkan graphical program can be a bit intimidating initially due to the amount of code and details (1000+ lines). Hence, to make setup/api programming aspect digestible and easier to debug, you'll subdivided your implementation into a set of self-contained functions (see Listing 6.1) as presented by Kenwright [8]. The implementation is 'functional' so variables are passed around, while and returned data/values are stored and used in subsequent methods. This way you avoid globals while learning and analysing the reasons behind the API/graphical concepts.

If you're completely new to the Vulkan API - manually typing in the code samples instead of just running the working program will help you absorb and understanding the principles better (more time consuming but aids in deep learning the subject).

Listing 6.1: Steps to initializing and running a basic Vulkan graphical application (split into 11 easy to understand functional stages). As you master each element, you'll expand and customize the implementation to deepen your understanding. Each of the steps is explain in detail in subsequent sections.

```
void main(int argc, char** argv)
{
    // Step 1 - Initializing the Window
    int             width               = 800;
    int             height              = 600;
    HWND            windowHandle        = NULL;
    SetupWindow(width, height, &windowHandle);

    // Step 2 - Initialize Vulkan       (Section 6.1)
    VkInstance      instance            = NULL;
    VkSurfaceKHR    surface             = NULL;
    SetupVulkanInstance(windowHandle,
                        &instance,
                        &surface);

    // Step 3 - Find/Create Device and  (Section 6.3)
```

```
    //          Set-up your selected device
    VkPhysicalDevice physicalDevice     = NULL;
    VkDevice         device             = NULL;
    SetupPhysicalDevice(instance,
                        &physicalDevice,
                        &device);


    // Step 4 - Initialize Swap-Chain   (Section 6.4)
    VkSwapchainKHR   swapChain          = NULL;
    VkImage*         presentImages      = NULL;
    VkImageView*     presentImageViews  = NULL;
    SetupSwapChain( device,
                    physicalDevice,
                    surface,
                    &width,
                    &height,
                    &swapChain,
                    &presentImages,
                    &presentImageViews);

    // Step 5 - Create Render Pass      (Section 6.5)
    VkRenderPass     renderPass         = NULL;
    VkFramebuffer*   frameBuffers       = NULL;
    SetupRenderPass(device,
                    physicalDevice,
                    width,
                    height,
                    presentImageViews,
                    &renderPass,
                    &frameBuffers);

    // Step 6 - Create Command Pool/Buffer (Section 6.6)
    VkCommandBuffer  commandBuffer      = NULL;
    SetupCommandBuffer(device,
                       physicalDevice,
                       &commandBuffer);

    // Step 7 - Vertex Data/Buffer     (Section 6.9)
    VkBuffer         vertexInputBuffer = NULL;
    int              vertexSize        = 0;
    int              numberOfTriangles = 0;
    SetupVertexBuffer(device,
                      physicalDevice,
                      &vertexSize,
                      &numberOfTriangles,
                      &vertexInputBuffer );

    // Step 8 - Load/Setup Shaders     (Section 6.10)
    VkShaderModule   vertShaderModule  = NULL;
    VkShaderModule   fragShaderModule  = NULL;
    VkBuffer         buffer            = NULL;
    VkDeviceMemory   memory            = NULL;
    SetupShaderandUniforms(device,
                           physicalDevice,
                           width,
                           height,
                           &vertShaderModule,
                           &fragShaderModule,
                           &buffer,
                           &memory);
```

```

    // Step 9 - Setup Descriptors/Sets          (Section 6.12)
    VkDescriptorSet       descriptorSet         = NULL;
    VkDescriptorSetLayout descriptorSetLayout   = NULL;
    SetupDescriptors(device,
                     buffer,
                     &descriptorSet,
                     &descriptorSetLayout);

    // Step 10 - Pipeline                  (Section 6.13)
    VkPipeline       pipeline          = NULL;
    VkPipelineLayout pipelineLayout    = NULL;
    SetupPipeline( device,
                   width,
                   height,
                   vertexSize,
                   descriptorSetLayout,
                   vertShaderModule,
                   fragShaderModule,
                   renderPass,
                   &pipeline,
                   &pipelineLayout);

    // Step 11 - Render Loop               (Section 6.15)
    MSG msg;
    while( true )
    {
        // Continually force the window to be redrawn as long as no other Win32
        // messages are pending.
        PeekMessage( &msg, NULL, NULL, NULL, PM_REMOVE );
        if( msg.message == WM_QUIT ) break;

        // Your Window's applications is responsible for retrieving and
        // dispatching input messages to the window GUI in the main message loop
        TranslateMessage( &msg );
        DispatchMessage( &msg );

        // Render the screen
        RenderLoop( device,
                    width,
                    height,
                    numberOfTriangles,
                    swapChain,
                    commandBuffer,
                    presentImages,
                    frameBuffers,
                    renderPass,
                    vertexInputBuffer,
                    descriptorSet,
                    pipelineLayout,
                    pipeline);
    }// End while(..)
    return 0;

}// End WinMain(..)
```

6.1 *(Step 1 & 2) Initializing Vulkan (Instance Creation)*

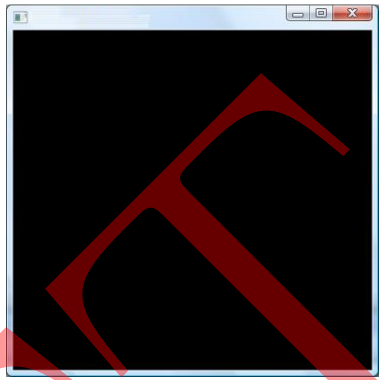

Figure 6.2: Uncomplicated window displaying using the Windows API (Step 1 - Initializing the Window).

The initial step is to setup your window for your operating system. As you have to let Vulkan where you're going to draw to (e.g., screen or off-screen texture). The OS specific parts of the implementation will be done for Windows, however, it should be straightforward to modify these few occurrences for different systems (e.g., Android and Linux).

The first Vulkan specific step you'll need to do after setting up your window is to initialize Vulkan. This is subdivided into two main parts. To begin with, you have to identify the Vulkan driver and characteristics you want to enable (e.g., standard LUNARG driver or the NVidia one, also what layers are available). For example, in the below implementation, you'll use 'vkEnumerateInstanceLayerProperties' and 'vkEnumerateInstanceExtensionProperties' to list all the layers and the instance properties. For the example, in the example implementation below, 'VK_LAYER_NV_optimus' has been hardcoded as the layer. Typically, you'll then also have three extensions, one will be the debug extension ('VK_EXT_debug_report'), which you'll include if you want to initialize the error callback notifications (discuss next). The next two extensions will depend upon your operating system and what you're using Vulkan for (e.g., graphics, compute, ...).

Just to note, in the Listing examples in subsequent sections, you'll often encounter additional 'curly brackets' .. inside the functions. These additional curly brackets have been added to clump together blocks of code so it's easier to read (i.e., self-contained modular components).

Listing 6.2: Initializing Vulkan - Vulkan doesn't exist until you create an the Vulkan instance (vkCreateInstance).

```
// Step 2 - Initialize Vulkan
void SetupVulkanInstance(HWND              windowHandle,
                         VkInstance*       outInstance,
                         VkSurfaceKHR*     outSurface)
{
  // Initialize VULKAN

  // Layer properties
  uint32_t count = 0;

  // Returns the number of layer properties available, If the VkLayerProperties*
  // is NULL, then the number of layer properties available is returned
  VkResult result =
  vkEnumerateInstanceLayerProperties(&count, // uint32_t*
                                      // pointer to the number of layer properties available
                                      NULL);  // VkLayerProperties*
                                      // pointer to an array of VkLayerProperties structures


  DBG_ASSERT( VK_SUCCESS == result );
```

```
DBG_ASSERT( count > 0 );

vector<VkLayerProperties> instanceLayers;
instanceLayers.resize(count);
// As the VkLayerProperties structure array is not NULL the function returns
// the layer properties
result =
vkEnumerateInstanceLayerProperties(&count, // uint32_t*
                                           // pointer to the number of layer properties available
                                           &instanceLayers[0]); // VkLayerProperties*          B
                                           // pointer to an array of VkLayerProperties structures



// Extension properties
// vkEnumerateInstanceExtensionProperties - Returns the requested number
// of global extension properties.  The count relates to the number of
// extension properties available
result =
vkEnumerateInstanceExtensionProperties(NULL, // const char*
                                           // pointer to the name of the layer to retrieve extensions for
                                           &count, // uint32_t*
                                           // pointer to the number of extension properties available
                                           NULL);  // VkExtensionProperties*          C
                                           // pointer to an array of VkExtensionProperties structures

DBG_ASSERT(VK_SUCCESS==result);
DBG_ASSERT(count > 0);

vector<VkExtensionProperties> instanceExtension;
instanceExtension.resize(count);
// Array of LayerNames not  NULL so returns an array of null-terminated UTF-8 strings
// names for the retrievable extensions.
// The VkExtensionProperties structures is not NULL so returns the extension properties
result =
vkEnumerateInstanceExtensionProperties(NULL,                // const char*
                                           // pointer to the name of the layer to retrieve extensions for
                                           &count,               // uint32_t*
                                           // pointer to the number of extension properties available
                                           &instanceExtension[0]); // VkExtensionProperties*          D
                                           // pointer to an array of VkExtensionProperties structures

vector<string> extensionNames;
extensionNames.resize(count);


/*
  e.g., VK_LAYER_NV_optimus or VK_LAYER_LUNARG_standard_validation
*/

const char *layers[] = { "VK_LAYER_NV_optimus" };

#ifdef ENABLE_VULKAN_DEBUG_CALLBACK // access debug callbacks
const char *extensions[] = { "VK_KHR_surface",
                             "VK_KHR_win32_surface",
                             "VK_EXT_debug_report"};

#else
const char *extensions[] = { "VK_KHR_surface",
                             "VK_KHR_win32_surface" };
```

```
#endif

  {
  VkApplicationInfo ai         = { };
  ai.sType                     = VK_STRUCTURE_TYPE_APPLICATION_INFO;
  ai.pApplicationName          = "Hello Vulkan";
  ai.engineVersion             = 1;
  ai.apiVersion                = VK_API_VERSION_1_0;

  VkInstanceCreateInfo ici     = { };
  ici.sType                    = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
  ici.flags                    = 0;
  ici.pNext                    = 0;
  ici.pApplicationInfo         = &ai;
  ici.enabledLayerCount        = 1;
  ici.ppEnabledLayerNames      = layers;
  ici.enabledExtensionCount    = 2;
#ifdef ENABLE_VULKAN_DEBUG_CALLBACK // access debug callbacks
  ici.enabledExtensionCount    = 3;
#endif
  ici.ppEnabledExtensionNames = extensions;

  // vkCreateInstance verifies that the requested layers exist. If not,
  // vkCreateInstance will return VK_ERROR_LAYER_NOT_PRESENT
  VkResult result =
  vkCreateInstance( &ici,         // const VkInstanceCreateInfo*
                       // points to an instance of VkInstanceCreateInfo controlling creation
                       NULL,         // const VkAllocationCallbacks*
                       // controls host memory allocation
                       outInstance ); // VkInstance*
                       // pointer to a VkInstance handle for the returning resulting instance

  DBG_ASSERT_VULKAN_MSG( result,
    "Failed to create vulkan instance." );
  DBG_ASSERT(*outInstance!=NULL);
  }

  // Optional - if you want Vulkan to tell you if something is wrong
  // you must set the callback
#ifdef ENABLE_VULKAN_DEBUG_CALLBACK
  ...
#endif


  // You need to define what type of surface you'll be
  // rendering to - this will depend on your computer
  // and operating system (Win32)
  HINSTANCE hInst                    = GetModuleHandle(NULL);

  // setup parameters for your new windows
  // surface you'll render into:
  VkWin32SurfaceCreateInfoKHR sci = {};
  sci.sType                          = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
  // parameter is NULL, GetModuleHandle returns a handle
  // to the file used to create the calling process
  sci.hinstance                      = hInst;
  // Your window handle (HWND)
  sci.hwnd                           = windowHandle;

  DBG_ASSERT(*outSurface==NULL);

```

```
// To create a VkSurfaceKHR object for a Win32 window, call:
VkResult result =
 vkCreateWin32SurfaceKHR ( *outInstance, // VkInstance
                            //  instance to associate with the surface
                            &sci,        // const VkWin32SurfaceCreateInfoKHR*
                            // pointer to VkWin32SurfaceCreateInfoKHR structure parameters for the surface object
                            NULL,        // const VkAllocationCallbacks*
                            // allocator used for host memory allocated for the surface object
                            outSurface ); // VkSurfaceKHR*                                          F
                            // VkSurfaceKHR handle in which the created surface object is returned

 DBG_ASSERT_VULKAN_MSG( result,
   "Could not create surface." );
 DBG_ASSERT(outSurface!=NULL);

}// End SetupVulkanInstance(..)
```

The process for setting up the Vulkan instance is:

A identify the available layers and extensions (e.g., vkEnumerateInstanceLayerProperties)

E create the Vulkan instance (completing the structure parameters for all the information, such as, version, name, ..)

F create the output surface and connect it with the operating system specific window (Window handle in this case)

While Listing 6.2, focuses on a Microsoft Windows solution, similar functions are available for platform specific Vulkan API (e.g., vkCreateWin32SurfaceKHR), such as, Android:

```
// To create a VkSurfaceKHR object for an Android native window, you'd call:
VkResult vkCreateAndroidSurfaceKHR(
          VkInstance                              instance,
          const VkAndroidSurfaceCreateInfoKHR*    pCreateInfo,
          const VkAllocationCallbacks*            pAllocator,
          VkSurfaceKHR*                           pSurface);
```

6.2 *Debugging*

You should start thinking about debugging (and defensive programming) from the start. For instance, a few reasons debugging in Vulkan is challenging:

- May be no obvious relationship between the manifestation(s) of the error and the causes(s)
- Symptoms and cause may be in remote/different parts of the program
- Changes (new features and bug fixes) in the program may mask (or modify) bugs

- Symptoms may be due to human mistakes or misunderstanding that is difficult to trace
- Bugs may be triggered by rare or difficult to reproduce sequences, program timing (threads) or other causes
- Bugs may depend on other software/system states (external libraries/code)

The default Vulkan API does not enable debugging/checking. Hence, you need to link in to the debug report callback functions to provide you with feedback on warning and issues as they occur.

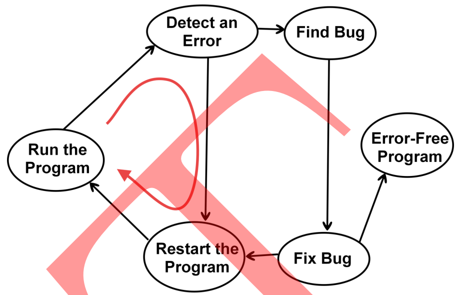

Figure 6.3: Conventional debugging cycle.

Listing 6.3: Enabling the built in debugging and warning feedback notifications within Vulkan.

```
// Optional - if you want Vulkan to tell you if something is wrong
// you must set the callback
#ifdef ENABLE_VULKAN_DEBUG_CALLBACK
{
  // Register your error logging function (defined at the top of the file)
  VkDebugReportCallbackEXT error_callback   = VK_NULL_HANDLE;
  VkDebugReportCallbackEXT warning_callback = VK_NULL_HANDLE;

  PFN_vkCreateDebugReportCallbackEXT vkCreateDebugReportCallbackEXT = NULL;      A

  *(void **)&vkCreateDebugReportCallbackEXT =
     vkGetInstanceProcAddr( *outInstance, "vkCreateDebugReportCallbackEXT" );
  DBG_ASSERT(vkCreateDebugReportCallbackEXT);                                      B


  VkDebugReportCallbackCreateInfoEXT cb_create_info = {};
  cb_create_info.sType       = VK_STRUCTURE_TYPE_DEBUG_REPORT_CREATE_INFO_EXT;
  cb_create_info.flags       = VK_DEBUG_REPORT_ERROR_BIT_EXT;
  cb_create_info.pfnCallback = &MyDebugReportCallback;

  // Setup error callback function notifications
  VkResult result =
  vkCreateDebugReportCallbackEXT(*outInstance,
                                         // valid VkInstance handle
                                         &cb_create_info,
                                         // // pointer to a valid VkDebugReportCallbackCreateInfoEXT structure
                                         nullptr,
                                         // If pointer is not NULL then allocator callback manager
                                         &error_callback);                         C
                                         // pointer to a VkDebugReportCallbackEXT handle

  DBG_ASSERT_VULKAN_MSG(result, "vkCreateDebugReportCallbackEXT(ERROR) failed");

  // Capture warning as well as errors
  cb_create_info.flags = VK_DEBUG_REPORT_WARNING_BIT_EXT |
          VK_DEBUG_REPORT_PERFORMANCE_WARNING_BIT_EXT;
  cb_create_info.pfnCallback = &MyDebugReportCallback;

  // Setup warning callback function notifications
  result =
  vkCreateDebugReportCallbackEXT(*outInstance,
                                         // valid VkInstance handle
                                         &cb_create_info,
```

```
                                                  // pointer to a valid VkDebugReportCallbackCreateInfoEXT structure
                                                  nullptr,
                                                  // If pointer is not NULL then allocator callback manager
                                                  &warning_callback);
                                                  // pointer to a VkDebugReportCallbackEXT handle

    DBG_ASSERT_VULKAN_MSG(result, "vkCreateDebugReportCallbackEXT(WARN) failed");
  }
#endif
```

A good habit to get into is using regular sanity checks throughout your implementation. For example, debug asserts (DBG_ASSERT) as shown below. The Vulkan API requires a large number of structures and fields to be setup and configured. For any unknown reason, such as, a typing mistake or some custom detail specific to the hardware, may result in the graphical application failing - importantly, leaving you struggling to work out why. Hence, try and check every return value (e.g., 'VK_SUCCESS') and if a function fails - trigger an assert (don't try and hide the problem) - have the error shout out with the details so you can investigate why it failed and resolve the problem asap. This is also a good habit to get into for helping others, as it makes your code more readable - so other developers are able to understand your code easier and if it fails they're also able to fix the problem quickly as well.

Listing 6.4: Custom asserts to provide an additional layer of checking. Custom asserts also provide flexibility (e.g., write to log files, trigger breakpoints, or disable them easily).

```
#if defined(_WIN32)
  #define DBG_ASSERT(f) { if(!(f)){ __debugbreak(); }; }
#else
  #define DBG_ASSERT(f) { } // Other operating system debug
#endif

// NAN Test
#define DBG_VALID(f)  { if( (f)!=(f) ){ DBG_ASSERT(false);} }

// Assert with message
#define DBG_ASSERT_MSG( val, errmsg )          \
    if(!(val)){                                \
       DBG_ASSERT( false )                     \
    }

#define DBG_ASSERT_VULKAN_MSG( val, errmsg )   \
    if(!((VK_SUCCESS == val))){                \
       DBG_ASSERT( false )                     \
    }
```

While in the long run, you'd incorporate a variety of complex test functions within a structured framework (unit tests), yet asserts provide an effective and efficient debugging tool for identifying issues during the initial phases. You need to use a custom 'macro' instead of the system assert directly, so you're able to control your asserts - like having the assert trigger a breakpoint at the line causing the validation fault. Furthermore, for release builds, you'd be able to customize the macro so instead of 'triggering' a breakpoint, you may want to write the error to a log file or bring up a dialog error window.

6.3 *(Step 3) Device(s)*

The system may have multiple devices. Each device may have similar or different capabilities. The physical device is identified in Vulkan using the type 'VkPhysicalDevice. Provides a handle to query the device about its capabilities, such as, Memory Management Queues Objects Buffers Images and Sync Primitives. For example, the 'Geforce GTX 980' has different capabilities than the 'Tegra X1'.

Figure 6.4: The Vulkan API is designed to support 'multiple' devices with varying capabilities.

Listing 6.5: Determining what devices are on your system and with what capabilities.

```
// Step 3 - Find/Create Device
void SetupPhysicalDevice(VkInstance         instance,
                         VkPhysicalDevice*  outPhysicalDevice,
                         VkDevice*          outDevice)
{
  // Query how many devices are present in the system
  uint32_t deviceCount = 0;
  // Enumerates the physical devices accessible to a Vulkan instance
  // The instance is the handle to a Vulkan instance you previously
  // created with vkCreateInstance.  The VkPhysicalDevice pointer
  // can be either NULL or a pointer to an array of VkPhysicalDevice handles.
  VkResult result =
  vkEnumeratePhysicalDevices(instance,      // VkInstance
                                   // handle to a Vulkan instance previously created with vkCreateInstance
                                   &deviceCount, // uint32_t*
                                   // pointer to an integer related to the number of physical devices available or queried
                                   NULL);        // VkPhysicalDevice*
                                   // either NULL or a pointer to an array of VkPhysicalDevice handles

  DBG_ASSERT_VULKAN_MSG(result,
    "Failed to query the number of physical devices present");

  // There has to be at least one device present
  DBG_ASSERT_MSG(0 != deviceCount,
    "Couldn't detect any device present with Vulkan support");

  // Get the physical devices
  vector<VkPhysicalDevice> physicalDevices(deviceCount);

  // Gets the VkPhysicalDevice handles.
  result =
  vkEnumeratePhysicalDevices(instance,             // VkInstance
                                   // handle to a Vulkan instance previously created with vkCreateInstance
                                   &deviceCount,         // uint32_t*
                                   // pointer to an integer related to the number of physical devices available or queried
                                   &physicalDevices[0]); // VkPhysicalDevice*
                                   // either NULL or a pointer to an array of VkPhysicalDevice handles
```

```
DBG_ASSERT_VULKAN_MSG(result,
  "Faied to enumerate physical devices present");
DBG_ASSERT(physicalDevices.size()>0);

// Use the first available device
*outPhysicalDevice = physicalDevices[0];


// Enumerate all physical devices and print out the details
for (uint32_t i = 0; i < deviceCount; ++i)
{
  VkPhysicalDeviceProperties deviceProperties;
  memset(&deviceProperties, 0, sizeof deviceProperties);

  // Gets the properties of a physical device
  vkGetPhysicalDeviceProperties(physicalDevices[i], // physicalDevice
                                      // handle to the physical device whose properties will be queried
                                      &deviceProperties); // pProperties
                                      // pointer to VkPhysicalDeviceProperties structure, that is filled with information

  dprintf("Driver Version: %d\n", deviceProperties.driverVersion);
  dprintf("Device Name:    %s\n", deviceProperties.deviceName);
  dprintf("Device Type:    %d\n", deviceProperties.deviceType);
  dprintf("API Version:    %d.%d.%d\n",
   (deviceProperties.apiVersion>>22)&0x3FF,
   (deviceProperties.apiVersion>>12)&0x3FF,
   (deviceProperties.apiVersion&0xFFF));
}//End for i


// Fill up the physical device memory properties:
VkPhysicalDeviceMemoryProperties memoryProperties;
vkGetPhysicalDeviceMemoryProperties( *outPhysicalDevice,
                                      // handle to the device to query
                                      &memoryProperties );
                                      // pointer to VkPhysicalDeviceMemoryProperties structure returned with properties
// Here's where you initialize your queues
// You'll discuss queues next - however, you need to specify the queue
// details for the device creation info
VkDeviceQueueCreateInfo queueCreateInfo = {};
queueCreateInfo.sType             = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
// Use the first queue family in the family list
queueCreateInfo.queueFamilyIndex  = 0;
queueCreateInfo.queueCount        = 1;
float queuePriorities[]           = { 1.0f };
queueCreateInfo.pQueuePriorities  = queuePriorities;

// Same extension you specified when initializing Vulkan
const char *deviceExtensions[]    = { "VK_KHR_swapchain" };

const char *layers[]              = { "VK_LAYER_NV_optimus" };

VkDeviceCreateInfo dci            = {};
dci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
// Set queue info on your device
dci.queueCreateInfoCount          = 1;
dci.pQueueCreateInfos             = &queueCreateInfo;
dci.enabledLayerCount             = 0;
dci.ppEnabledLayerNames           = layers;
dci.enabledExtensionCount         = 1;
```

```
dci.ppEnabledExtensionNames       = deviceExtensions;

VkPhysicalDeviceFeatures features = {};
features.shaderClipDistance       = VK_TRUE;
dci.pEnabledFeatures              = &features;

// Ideally, you'd want to enumerate and find the best
// device, however, you just use the first device
// 'physicalDevices[0]' for your sample, which you
// stored in the previous section
result =
 vkCreateDevice ( *outPhysicalDevice, // physicalDevice
                  // valid handles returned from vkEnumeratePhysicalDevices
                  &dci,              // pCreateInfo
                  // pointer to a VkDeviceCreateInfo structure containing device data
                  NULL,              // pAllocator
                  // optional control of host memory allocation
                  outDevice );       // pDevice
                  // pointer to a handle in which the created VkDevice is returned

DBG_ASSERT_VULKAN_MSG( result, "Failed to create logical device!" );
}// End SetupPhysicalDevice(..)
```

With reference to Listing 6.5, the flow of logic to finding and creating the physical device are:

- A vkEnumeratePhysicalDevices - query how many devices are present in the system
- B vkEnumeratePhysicalDevices - call again to get the physical devices
- C vkGetPhysicalDeviceProperties - get properties for each device (help make your decision on which one to choose or on the selected one)
- D vkGetPhysicalDeviceMemoryProperties - more properties on the chosen device before you go ahead and create the device
- E vkCreateDevice - finally create your device

6.4 *(Step 4) Swap-Chain*

There is 'no' default framebuffer in Vulkan. You are able to create an application that displays everything or nothing (total control). Hence, to display something you'll need to create a set of render buffers. These buffers (and their properties) are called the 'swap chain'. As emphasised, you have total control over your swap chain, which means, you can create and use lots of buffers however you want. A few important details when creating your swap chain image buffers:

1. define the surface format
2. create rendering context (connect the swap chain with the presentation output)

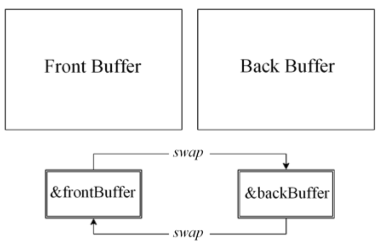

Figure 6.5: The swap-chain is a chain of buffers that swap position each time a new frame is rendered.

3. you'll need to be able to 'destroy' and 'recreate' the swap chain if the window or parameters change (e.g., window resized or the user changes the render options)

Listing 6.6: Managing the screen capabilities and render surfaces.

```
// Step -4-
void SetupSwapChain(VkDevice          device,
                    VkPhysicalDevice  physicalDevice,
                    VkSurfaceKHR      surface,
                    int*              outWidth,
                    int*              outHeight,
                    VkSwapchainKHR*   outSwapChain,
                    VkImage**         outPresentImages,
                    VkImageView**     outPresentImageViews)
{
  {
  // Create swap-chain
  // swap-chain creation:
  VkSurfaceCapabilitiesKHR surfaceCapabilities = {};
  // You'll query the basic capabilities of the surface in order to create a swapchain
  vkGetPhysicalDeviceSurfaceCapabilitiesKHR(physicalDevice,        // physicalDevice
                                                       // physical device that will be associated with the swapchain to be created
                                                       surface,              // surface
                                                       // surface that will be associated with the swapchain
                                                       &surfaceCapabilities); // pSurfaceCapabilities
                                                       // pointer to the VkSurfaceCapabilitiesKHR structure with retrived data

  VkExtent2D surfaceResolution  = surfaceCapabilities.currentExtent;
  *outWidth                     = surfaceResolution.width;
  *outHeight                    = surfaceResolution.height;


  VkSwapchainCreateInfoKHR ssci = {};
  ssci.sType                    = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
  ssci.surface                  = surface;
  // You'll use 2 for 'double' buffering
  ssci.minImageCount            = 2;
  ssci.imageFormat              = VK_FORMAT_B8G8R8A8_UNORM;
  ssci.imageColorSpace          = VK_COLORSPACE_SRGB_NONLINEAR_KHR;
  ssci.imageExtent              = surfaceResolution;
  ssci.imageArrayLayers         = 1;
  ssci.imageUsage               = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
  ssci.imageSharingMode         = VK_SHARING_MODE_EXCLUSIVE;
  ssci.preTransform             = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
  ssci.compositeAlpha           = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
  ssci.presentMode              = VK_PRESENT_MODE_MAILBOX_KHR;
  // If you want clipping outside the extents
  ssci.clipped                  = true;
  ssci.oldSwapchain             = NULL;

  VkResult result =
  vkCreateSwapchainKHR( device,         // device
                        // VkDevice to associate the swapchain
                        &ssci,          // pCreateInfo
                        // pointer to VkSwapchainCreateInfoKHR structure with swapchain creation parameters
                        NULL,           // pAllocator
                        // optional allocator used for host memory
                        outSwapChain ); // pSwapchain
```

```
                                  // resulting swapchain
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create swapchain." );
}

// Create your images 'double' buffering
{
uint32_t imageCount    = 0;
// You'll need to obtain the array of presentable images associated
// with the swapchain you created.  First, you pass in 'NULL' to
// obtain the number of images (i.e., should be 2)
vkGetSwapchainImagesKHR( device,        // device
                                      // device associated with swapchain
                                      *outSwapChain, // swapchain
                                      // swapchain to query
                                      &imageCount,   // pSwapchainImageCount
                                      // pointer to an integer related to the number of format pairs available
                                      NULL );        // pSwapchainImages
                                      // either NULL or a pointer to an array of VkSwapchainImageKHR structures
DBG_ASSERT(imageCount==2);

// this should be 2 for double-buffering
*outPresentImages = new VkImage[imageCount];

// Obtain the presentable images and link them to
// the images in the swapchain
VkResult result =
vkGetSwapchainImagesKHR( device,             // device
                                      // device associated with swapchain
                                      *outSwapChain,      // swapchain
                                      // swapchain to query
                                      &imageCount,        // pSwapchainImageCount
                                      // pointer to an integer related to the number of format pairs available
                                      *outPresentImages); // pSwapchainImages
                                      // either NULL or a pointer to an array of VkSwapchainImageKHR structures

DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create swap-chain images" );
}


{
// You have 2 for double buffering
*outPresentImageViews = new VkImageView[2];
for( uint32_t i = 0; i < 2; ++i )
{
// create VkImageViews for your swap chain
// VkImages buffers:
VkImageViewCreateInfo ivci              = {};
ivci.sType                              = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
ivci.viewType                           = VK_IMAGE_VIEW_TYPE_2D;
ivci.format                             = VK_FORMAT_B8G8R8A8_UNORM;
ivci.components.r                       = VK_COMPONENT_SWIZZLE_R;
ivci.components.g                       = VK_COMPONENT_SWIZZLE_G;
ivci.components.b                       = VK_COMPONENT_SWIZZLE_B;
ivci.components.a                       = VK_COMPONENT_SWIZZLE_A;
ivci.subresourceRange.aspectMask        = VK_IMAGE_ASPECT_COLOR_BIT;
ivci.subresourceRange.baseMipLevel      = 0;
ivci.subresourceRange.levelCount        = 1;
ivci.subresourceRange.baseArrayLayer    = 0;
```

```
  ivci.subresourceRange.layerCount      = 1;
  ivci.image                            = (*outPresentImages)[i];

  // Create an image view from an existing image
  VkResult result =
  vkCreateImageView( device,                         // device
                         // logical device that creates the image view
                         &ivci,                          // pCreateInfo
                         // pointer to instance of the VkImageViewCreateInfo structure containing parameters for the image view
                         NULL,                           // pAllocator
                         // optional controls host memory allocation
                         &(*outPresentImageViews)[i] ); // pView
                         // pointer to VkImageView handle for returned image view object

  DBG_ASSERT_VULKAN_MSG( result,
    "Could not create ImageView." );
  }// End for i
  }
}// End SetupSwapChain(..)
```

Looking at Listing 6.6, you'll see the implementation specifics for configuring and setting up your swapchain:

- A vkGetPhysicalDeviceSurfaceCapabilitiesKHR
- B vkCreateSwapchainKHR
- C vkGetSwapchainImagesKHR is called twice as you'll want to double buffer the swap chain (front and back buffer)
- E vkCreateImageView

6.5 *(Step 5) FrameBuffer & Render-Pass*

The framebuffer in Vulkan is simpler than previous traditional OpenGL implementations. In Vulkan you have a 'Bag' or 'Repository' of resource views. The render-pass defines the role of framebuffer resources. Importantly, you can have more than one pass with each pass defining which framebuffer resource to use. While the render-pass might seem like additional work, as you start to generate more complex scenes, the render-pass gives you additional screen control. For example, post-processing and deferred rendering (e.g., mapping specific regions or order of processing to different threads/GPUs). The listing below sets a basic fullscreen render-pass (i.e., one display update with no sub-passes). With reference to the command-buffer (in the next section), you can use the command-buffer for several render-passes. You can also use a single command-buffer to draw a whole frames with the multiple passes contributing to techniques like shadow mapping and post-processing (managing these process more efficiently).

```
// Step -5-
void SetupRenderPass(VkDevice          device,
                     VkPhysicalDevice  physicalDevice,
                     int               width,
                     int               height,
                     VkImageView*      presentImageViews,
                     VkRenderPass*     outRenderPass,
                     VkFramebuffer**   outFrameBuffers)
{
  // Frame buffer
  // define your attachment points

#ifdef DEPTH_BUFFER
  // Extension (Depth Buffer)
  VkImage         depthImage          = NULL;
  VkImageView     depthImageView      = NULL;

  {
  // create a depth image:
  VkImageCreateInfo imageCreateInfo     = {};
  imageCreateInfo.sType                 = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
  imageCreateInfo.imageType             = VK_IMAGE_TYPE_2D;
  imageCreateInfo.format                = VK_FORMAT_D16_UNORM;
  VkExtent3D ef                         = { width, height, 1 };
  imageCreateInfo.extent                = ef;
  imageCreateInfo.mipLevels             = 1;
  imageCreateInfo.arrayLayers           = 1;
  imageCreateInfo.samples               = VK_SAMPLE_COUNT_1_BIT;
  imageCreateInfo.tiling                = VK_IMAGE_TILING_OPTIMAL;
  imageCreateInfo.usage                 = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
  imageCreateInfo.sharingMode           = VK_SHARING_MODE_EXCLUSIVE;
  imageCreateInfo.queueFamilyIndexCount = 0;
  imageCreateInfo.pQueueFamilyIndices   = NULL;
  imageCreateInfo.initialLayout         = VK_IMAGE_LAYOUT_UNDEFINED;

  // Create a new image object for your depth buffer
  VkResult result =
  vkCreateImage( device,           // device
                     // logical device that creates the image
                     &imageCreateInfo, // pCreateInfo
                     // pointer to VkImageCreateInfo structure with parameters for the created image
                     NULL,             // pAllocator
                     // optional control host memory allocation
                     &depthImage );    // pImage                                                A
                     // pointer to VkImage handle returned image object

  DBG_ASSERT_VULKAN_MSG( result,
      "Failed to create depth image." );

  VkMemoryRequirements memoryRequirements = {};
  vkGetImageMemoryRequirements( device,              // device
                                 // logical device that owns the image
                                 depthImage,          // image
                                 // image to query
                                 &memoryRequirements ); // pMemoryRequirements                    B
                                 // instance pointer to VkMemoryRequirements structure returned memory requirements

  // memoryRequirements contains memoryTypeBits member which is a bitmask - each one of the
  // bits is set for every supported memory type for the resource. Bit i is set if and only
  // if the memory type i in the VkPhysicalDeviceMemoryProperties structure for the physical
```

```
// device is supported for the resource.

// Allocate memory for your depth buffer
VkMemoryAllocateInfo imageAllocateInfo  = {};
imageAllocateInfo.sType                 = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
imageAllocateInfo.allocationSize        = memoryRequirements.size;

// memoryTypeBits is a bitfield where if bit i is set, it means that
// the VkMemoryType i of the VkPhysicalDeviceMemoryProperties structure
// satisfies the memory requirements:
// read the device memory properties
VkPhysicalDeviceMemoryProperties memoryProperties;
vkGetPhysicalDeviceMemoryProperties( physicalDevice,
                                     // handle to the device to query.
                                     &memoryProperties );
                                     // returned pointer to instance of VkPhysicalDeviceMemoryProperties structure

uint32_t memoryTypeBits = memoryRequirements.memoryTypeBits;
for( uint32_t i = 0; i < VK_MAX_MEMORY_TYPES; ++i )
{
  VkMemoryType memoryType = memoryProperties.memoryTypes[i];
  if( memoryTypeBits & 1 )
  {
      if( ( memoryType.propertyFlags & VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT ) )
      {
          // save index
          imageAllocateInfo.memoryTypeIndex = i;
          break;
      }
  }
  memoryTypeBits = memoryTypeBits >> 1;
}// End for i

VkDeviceMemory imageMemory = { 0 };
result = vkAllocateMemory( device,
                           // logical device that owns the memory
                           &imageAllocateInfo,
                           // pointer to VkMemoryAllocateInfo structure describing parameters of the allocation
                           NULL,
                           // optional control host memory allocation
                           &imageMemory );
                           // pointer to returned VkDeviceMemory handle with information about the allocated memory

DBG_ASSERT_VULKAN_MSG( result, "Failed to allocate device memory." );

result = vkBindImageMemory( device,
                            // logical device that owns the image and memory
                            depthImage,
                            // image to bind
                            imageMemory, 0 );
                            // start offset of the region of memory which is to be bound to the image

DBG_ASSERT_VULKAN_MSG( result, "Failed to bind image memory." );

// create the depth image view:
VkImageAspectFlags aspectMask               = VK_IMAGE_ASPECT_DEPTH_BIT;
VkImageViewCreateInfo imageViewCreateInfo   = {};
imageViewCreateInfo.sType                   = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
imageViewCreateInfo.image                   = depthImage;
imageViewCreateInfo.viewType                = VK_IMAGE_VIEW_TYPE_2D;
```

```
  imageViewCreateInfo.format                          = imageCreateInfo.format;
  VkComponentMapping g                                = { VK_COMPONENT_SWIZZLE_IDENTITY,
                                                          VK_COMPONENT_SWIZZLE_IDENTITY,
                                                          VK_COMPONENT_SWIZZLE_IDENTITY,
                                                          VK_COMPONENT_SWIZZLE_IDENTITY };
  imageViewCreateInfo.components                      = g;
  imageViewCreateInfo.subresourceRange.aspectMask     = aspectMask;
  imageViewCreateInfo.subresourceRange.baseMipLevel   = 0;
  imageViewCreateInfo.subresourceRange.levelCount     = 1;
  imageViewCreateInfo.subresourceRange.baseArrayLayer = 0;
  imageViewCreateInfo.subresourceRange.layerCount     = 1;
  result =
  vkCreateImageView( device,
                        // logical device that creates the image view
                        &imageViewCreateInfo,
                        // pointer to instance of the VkImageViewCreateInfo structure containing parameters for created image view
                        NULL,
                        // optional control host memory allocation
                        &depthImageView );
                        // pointer to returned VkImageView handle object

  DBG_ASSERT_VULKAN_MSG( result,
    "Failed to create image view." );
  }
#endif // DEPTH_BUFFER

  // 0 - color screen buffer
  VkAttachmentDescription pass[2] = { };
  pass[0].format              = VK_FORMAT_B8G8R8A8_UNORM;
  pass[0].samples             = VK_SAMPLE_COUNT_1_BIT;
  pass[0].loadOp              = VK_ATTACHMENT_LOAD_OP_CLEAR;
  pass[0].storeOp             = VK_ATTACHMENT_STORE_OP_STORE;
  pass[0].stencilLoadOp       = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  pass[0].stencilStoreOp      = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  pass[0].initialLayout       = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
  pass[0].finalLayout         = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

  VkAttachmentReference car   = {};
  car.attachment              = 0;
  car.layout                  = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

  // 1 - depth buffer
  pass[1].format              = VK_FORMAT_D16_UNORM;
  pass[1].samples             = VK_SAMPLE_COUNT_1_BIT;
  pass[1].loadOp              = VK_ATTACHMENT_LOAD_OP_CLEAR;
  pass[1].storeOp             = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  pass[1].stencilLoadOp       = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  pass[1].stencilStoreOp      = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  pass[1].initialLayout       = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
  pass[1].finalLayout         = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

  // create the one main subpass of your renderpass:
  VkSubpassDescription subpass    = {};
  subpass.pipelineBindPoint       = VK_PIPELINE_BIND_POINT_GRAPHICS;
  subpass.colorAttachmentCount    = 1;
  subpass.pColorAttachments       = &car;
  subpass.pDepthStencilAttachment = NULL;

#ifdef DEPTH_BUFFER
  VkAttachmentReference dar       = {};
  dar.attachment                  = 1;
```

```
  dar.layout                        = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
  subpass.pDepthStencilAttachment = &dar;
#endif
  // create your main renderpass
  VkRenderPassCreateInfo rpci     = {};
  rpci.sType                        = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
  rpci.attachmentCount              = 1; // color
#ifdef DEPTH_BUFFER
  rpci.attachmentCount              = 2; // color and depth
#endif
  rpci.pAttachments                 = pass;
  rpci.subpassCount                 = 1;
  rpci.pSubpasses                   = &subpass;

  VkResult result =
  vkCreateRenderPass( device,
                        // logical device that creates the render pass
                        &rpci,
                        // pointer to VkRenderPassCreateInfo structure describing parameters of the render pass
                        NULL,
                        // optional host memory allocation control
                        outRenderPass );
                        // pointer VkRenderPass handle in which the resulting render pass object is returned

  DBG_ASSERT_VULKAN_MSG( result,
    "Failed to create renderpass" );

#ifdef DEPTH_BUFFER
  VkImageView frameBufferAttachments[2] = {0};
#else
  VkImageView frameBufferAttachments[1] = {0};
#endif
  // create your frame buffers:
  VkFramebufferCreateInfo fbci    = {};
  fbci.sType                        = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
  fbci.renderPass                   = *outRenderPass;
  // must be equal to the attachment count on render pass
  fbci.attachmentCount              = 1;
#ifdef DEPTH_BUFFER
  fbci.attachmentCount              = 2;
#endif
  fbci.pAttachments                 = frameBufferAttachments;
  fbci.width                        = width;
  fbci.height                       = height;
  fbci.layers                       = 1;


  // create a framebuffer per swap chain imageView:
  *outFrameBuffers = new VkFramebuffer[ 2 ];
  for( uint32_t i = 0; i < 2; ++i )
  {
    frameBufferAttachments[0] = presentImageViews[ i ];
#ifdef DEPTH_BUFFER
    frameBufferAttachments[1] = depthImageView;
#endif
    // Create a new framebuffer object
    result =
    vkCreateFramebuffer( device,              // device
                           // logical device that creates the framebuffer
                           &fbci,                 // pCreateInfo
```

```
                                   // pointer to VkFramebufferCreateInfo structure describing framebuffer creation
                                   NULL,                  // pAllocator
                                   // optional control of host memory allocation
                                   &(*outFrameBuffers)[i]); // pFramebuffer
                                   // pointer to returned VkFramebuffer handle for the framebuffer object

    DBG_ASSERT_VULKAN_MSG( result,
      "Failed to create framebuffer.");
  }// End for i
}// End SetupRenderPass(..)
```

Looking at Listing 6.5, you'll see the implementation specifics for configuring and setting up your framebuffer and render-pass:

- A vkCreateImage
- B vkGetImageMemoryRequirements
- C vkGetPhysicalDeviceMemoryProperties
- D vkAllocateMemory
- E vkBindImageMemory
- F vkCreateImageView
- G vkCreateRenderPass
- H vkCreateFramebuffer

6.6 *(Step 6) Command-Buffers*

Vulkan Rendering is done through Command-Buffers. The Command-Buffers are allocated from Command-Pools. Typically you have a Command-Pools associated with each thread and only use this thread when you write to the Command-Buffers allocated from its Command-Pool. This is because, it would be inefficient to externally synchronize access between the Command-Buffers and the Command-Pools (i.e., added overhead). Each Command-Buffer can be created either for one shot case or for multiple frames/submissions. Cannot call Command-Buffers from GPU (command-lists can). The API commands for filling the Command-Buffer begin with 'vkCmd'..() and need to be done between a 'Begin' and 'End'. Importantly, the Command-Buffer mechanism is designed to be multi-threading friendly. The 'primary' Command-Buffer can call many secondary Command-Buffers.

Listing 6.7: Command-Buffers are crucial elements for controlling the renderering.

```
// Step -6-
void SetupCommandBuffer(VkDevice          device,
                        VkPhysicalDevice  physicalDevice,
                        VkCommandBuffer*  outCommandBuffer)
{
  // Give your device some commands (orders)
  {
```