COM 391 SUMMARY

MORITZ M. KONARSKI

Contents

1. Introduction

Summary for the course Computer Graphics (COM 391) at the American University of Central Asia (https://www.auca.kg) in Bishkek, Kyrgyzstan for the spring semester of 2020. In this course we will write a 3D graphics engine in C++ using OpenGL.

2. History

In 1963 **Sketchpad** by Ivan Sutherland was one of the first CAD prototypes. In 1968 the **Mother of all demos** showed a cursor, graphical UI, real time collaboration on documents, and much more. The first semi-realistic shading algorithm was developed by Bui Tuong **Phong** at the University of Utah in 1973. In 1975 the **Utah teapot** was first used as a 3D rendering test model that has since been a popular in-joke. In 1973 the first computer with a window-based GUI by Xerox Alto shipped and in 1984 the first **Macintosh** shipped and **Tron**, the first heavily CGI based movie, was released. **Photoshop** was developed for that Mac in 1985. In the same year the **IRIS**

Date: March 30, 2020.

SGI Graphics Workstation was released. The 1995 movie **Toy Story** was the first entirely computer-animated feature film. **3D graphics accelerators** were becoming a thing with ATI, Nvidia, and 3dfx.

3. GPU Architecture

GPU dies are generally larger than CPU dies and they contain many more cores. Individual GPU cores are smaller than CPU cores though. In a CPU each core generally has its own L1 and L2 cache and then all cores share the L3 cache. In GPUs a cluster of cores shares L1 cache and clusters of clusters share L2 cache. Some CPU manufacturers are Intel, AMD, Qualcomm, IBM, and ARM. Some GPU manufacturers are Nvidia, AMD, Qualcomm, Intel, and ARM.

4. GPU APIs

Khronos Group maintains a number of Graphics **APIs** (application programming interfaces, a set of functions that allow the access to, in this case, GPU features). Those APIs are **OpenGL** (1, 2, 4, 4), OpenGL ES (2, 3), and Vulkan. Microsoft maintains **DirectX** (Direct3D), Apple maintains **Metal**, and AMD **Mantle**. Other APIs include CUDA, OpenCL, and OpenVG. While C, C++, Java etc are common CPU languages, HLSL, GLSL, and CUDA are common GPU languages. The programming languages used for APIs are generally C/C++ but other languages like Javascript, C#, Java, and Python can also be used.

Popular libraries used with these APIs are **SDL2** (Single DirectMedia Layer, a low level library for accessing OpenGL or Direct3D), **GLM** (OpenGL Mathematics, a library providing mathematical tools like vectors and matrices as well as their operations), and **GLEW** (OpenGL Extension Wrangler library is needed to determine the supported OpenGL features on a platform at runtime). Some graphics engines include Unity, Unreal Engine, CRYENGINE, Frostbite. Graphics engines and APIs run on basically all systems, be it PC, console, phone, TV or embedded. VR and AR are also current development areas.

5. Mathematical Basics

5.1. **Vectors.** Basically just an ordered list of numbers. The **dimension** of a vector is the number of elements in it. A vector has a magnitude (length) and a direction while a **scalar** only has a magnitude. In game development the most common vectors are 2D, 3D, and 4D vectors. Line or row vectors can be written like $\vec{v} = [1, 2, 3]$. Writing the same vector vertically gives a column vector. Scalar values are generally represented by lowercase letters (Greek or Roman) in italics: *a, b, c,* α, γ, θ while vectors are represented by bold lowercase letters **a, b, c, v** or by an arrow over top $\vec{a}$, $\vec{v}$. To refer to elements of a vector, subscript notation is used $\vec{v}_1$, $\vec{v}_2$ or $\vec{v}_x$, $\vec{v}_y$

Geometrically speaking a vector is an arrow that has a magnitude (length) and direction. The magnitude of a vector is always positive. The arrow part of a drawn vector is called head, the end tail. The numeric and graphic representations of vectors are

identical. Vectors do not have a position in some space, they only encode displacements of things relative to other things. The **zero vector** is a special vector which has 0s in all positions. Its magnitude is 0 and it is the only vector where this is the case. It is also the only vector that does not have a position. Because vectors describe displacements they can encode relative positions. If displacements are considered from the origin, they can be used to represent points.

5.1.1. *Mathematical Operations.* To **negate a vector** all of its elements need to be negated. This has the effect of flipping the direction of the vector.

Multiplying a vector by a scalar results in all of its elements being multiplied by the scalar. This means the vector is stretched or compressed.

To **add or subtract two vectors** from each other, their corresponding elements are added or subtracted. Geometrically that means putting the tail of one vector at the head of the other one.

To find a **displacement vector between two points** A and B one subtracts A from B: $\vec{d} = B - A$.

The **magnitude or length of a vector** is found by taking the square root of the sum of the squares of all its elements $||\vec{a}|| = \sqrt{x^2 + y^2 + z^2}$.

Unit vectors (normalized vectors) encode only the direction of a vector and have the length 1 $\vec{v}_0 = \vec{v}/||\vec{v}||$.

The **dot product** is a product of two vectors where the corresponding elements are multiplied together and the results then added together to get one number. $\vec{a} * \vec{b} = \vec{a}_1 * b_1 + \vec{a}_2 * b_2 + \vec{a}_3 * b_3$. It also has the definition $\vec{a} * \vec{b} = ||\vec{a}|| * ||\vec{b}|| * \cos\theta$, θ being the angle between the vectors. From we know that if the two vectors are perpendicular to each other, their dot product must be 0.

The **cross product** can only be applied in at least 3D, is not commutative and yields another vector. Specifically, $\vec{a} \times \vec{b}$ yields a vector that is perpendicular to both original vectors. $\vec{a} \times \vec{b} = [a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1]$. This has the further property that $||\vec{a} \times \vec{b}|| = ||\vec{a}|| * ||\vec{b}|| * \sin\theta$.

5.2. **Matrices.** Matrices are really important in linear algebra and they generally describe the relationships between two coordinate spaces. This is done by defining a computation that transforms vectors from one coordinate space to another. Matrices are rectangular grids that are made up of rows (horizontal) and columns (vertical). Matrices are defined by the number of rows and columns that they have, **n** rows by **m** columns as $n \times m$. Matrices are represented by bold capital letters **M, A, R**. To refer to specific elements in a matrix subscripts are used, first row (i) then column (j) as m_{ij}.

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

Square matrices are matrices that have the same number of rows and columns. The most common examples are 2D, 3D, and 4D examples.

Diagonal elements of a matrix are those where the number for row and column are the same m_{ii}. These elements for the diagonal of the matrix. All other elements are called non-diagonal.

A **diagonal matrix** has 0 in all but the diagonal. If the diagonal only contains 1s the matrix is called **identity matrix I**. The dimension is indicated by a subscript

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The identity matrix is important because it is the **multiplicative identity** element for matrices meaning that multiplying by it does not change a matrix $A \times I = I \times A = A$. Matrices with dimensions $1 \times m$ or $n \times 1$ can be seen as row or column vectors.

A square matrix can describe any linear transformation (transformation that preserves straight and parallel lines and has no translation). This means these matrices encode rotations and multiplications (stretching). A list of useful transformations include

- Rotation: matrix that rotates coordinate spaces or vectors. For example rotating then counterclockwise by θ by finding $R\vec{v}$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

- Scale: increase or decrease lengths of a vector or distances by multiplying $S\vec{v}$

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}.$$

- Orthographic Projection: a form of parallel projection that represents 3D objects in 2D space. A simple example is the projection onto the plane $z = 0$

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

- Reflection: reflects and may also rotate at the same time. Example: $\pi/2$ reflection

$$R = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Multiplying a matrix by one of the base vectors i, j, k yields the row corresponding to the base vector used. In other words, the first row of **M** contains the transformation of i and so on. We can thus interpret the rows of a matrix as the basis vectors of some coordinate system. If we understand how the basis vectors are changed by the rows of a matrix we can understand what the transformation does. Accordingly, to transform a vector from its original coordinate system to a new one we multiply it by the corresponding matrix. Thus a linear transformation is performed.

5.2.1. *Mathematical Operations.* **Matrix Transposition** is the process of flipping a matrix diagonally, so it turns $M_{nm} \to M^T_{mn}$. For vectors a transposition turns a row

vector into a column vector and vice versa. The notation can also be used to write column vectors in text like $[1,2,3]^T$. Double transpositions yield the original matrix. All diagonal matrices are equal to their transposed forms $D^T = D$.

Scalar multiplication for matrices $k * M$ multiplies all elements of M by k.

Matrix multiplication can be performed when the number of columns of the first matrix is equal to the number of rows of the second matrix. Multiplying a $r \times n$ matrix by a $n \times c$ matrix yields a $r \times c$ matrix. If the number of columns does not equal the number of rows the operation is not defined. Matrix multiplication is not commutative. It is defined for AB as each element c_{ij} of the result matrix as the dot product of row i of **A** with column j of **B**. If any matrix **M** is multiplied by a square matrix S of the right size (SM or MS) the resulting matrix has the same dimension as **M**. Matrix multiplication is not commutative but associative $(AB)C = A(BC)$. The transposition of a product is the same as the product of transpositions with order reversed $(AB)^T = B^T A^T$.

Multiplying a vector and matrix is possible if the number of columns equals the number of rows, both operations yield vectors. Row vector times matrix gives row vector, matrix times column vector gives column vector. Vector multiplication distributes over vector addition $(\vec{v} + \vec{w})M = \vec{v}M + \vec{w}M$

6. Book Notes

6.1. **Chapter 1.**

6.1.1. *What is 3D Math?*

- math behind the geometry of a 3D world
- solving geometric problems algorithmically
- book uses C++
- basically all the basics one needs for computations in 3D

6.1.2. *Why You Should Read This Book.*

- an introductory book
- a toolbox that one can come back to
- it has math, geometry, and code in one
- good graphics that illustrate 3D maths and concepts
- accompanying website with demos and stuff www.gamemath.com

6.1.3. *Chapter Overview.*

- 1. Introduction
- 2. Cartesian Coordinates
- 3. Examples of Coordinate Spaces and Nesting
- 4. Vectors
- 5. Vector Operations
- 6. C++ Vector Class
- 10. Angular Displacement

6.2. **Chapter 2.**

6.2.1. *1D Mathematics.*

- natural numbers / counting numbers, number line
- integers, negative numbers
- rational numbers, irrational numbers, real numbers
- complex numbers
- binary numbers and C++ number types
- **First Law of Computer Graphics**: If it looks right, it *is* right.

6.2.2. *2D Cartesian Mathematics.*

- cartesian = rectangular
- centered around the arbitrary origin
- infinite in all directions, continuous division of intervals
- normal orientation: positive y up, positive x right
- all of those orientations can be transformed into each other

6.2.3. *From 2D to 3D.*

- 3D tends to be a lot more complicated than 2D
- we have 3 perpendicular axes
- right handed is right hand with thumb pointed in +x, left handed same with thumb into +x

6.3. **Chapter 3.**

6.3.1. *Why Multiple Coordinate Spaces?*

- using more than one coordinate space tends to be easier even though one big world would be sufficient
- in some circumstances it makes sense to do certain things in separate coordinate systems and then bring them together when needed

6.3.2. *Some Useful Coordinate Spaces.*

- **World Space**: is the ultimate reference frame for all other coordinate systems; the largest coordinate system we care about; also know as **global** or **universal**; *position, orientation of objects (including camera), terrain elements, movement with respect to world space*
- **Object Space**: associated with each individual object; this object space is moved and rotated with the object when it moves and rotates in world space; for example *forward* is a concept of object space while *east* would be world space; *are other objects near, where is it in relation to me (front, left)*
- **Camera Space**: coordinate space associated with an observer; 3D space that is a special object space where the camera defines the viewpoint of the scene; this defines what *up* means with respect to the camera and not the world as a whole; *is a point in front of the camera, is a point on screen or off screen, where on the screen is it, objects in relation to each other*

- **Inertial Space**: coordinate space is half-way between world space and object space; origin is the same as of origin space; axes of this space are parallel to world space; this basically takes the *rotation* step out of the object to world space transition; leaves only the *translation* from world origin to object space origin

6.3.3. *Nested Coordinate Spaces.*

- an object is defined relative to the origin of object space
- the position and orientation of an object in world space is important for interactions with other objects
- the object's position can be figured out by positioning the origin in world space and then figuring out relative positions
- **world space is the parent of object space**
- objects can have child spaces and those can have child spaces themselves, this means coordinate spaces are nested

6.3.4. *Specifying Coordinate Spaces.*

- we need to specify an origin to specify the position
- the orientation is described by the coordinate axes

6.3.5. *Coordinate Space Transformations.*

- when we want to have two objects interact we need to express things in one coordinate system in terms of another one, e.g. a robot picking up an object where we know the position of the robot's hand in robot coordinates but need them in world coordinates
- we need to transform the position from world space to object space or vice versa – the transformation does not actually move anything, it just expresses it in different coordinates
- first we rotate the object axes to get them parallel to the world axes
- then we translate the inertial space to line it up with the world space

6.4. **Chapter 4.**

6.4.1. *Vector - Mathematical Definition.* See previous section on vectors.

6.4.2. *Vector - Geometric Definition.* See previous section on vectors.

- displacement and velocity are vector values while distance and speed are scalar quantities

6.5. **Chapter 5.**

6.5.1. *Linear Algebra.* See previous section on vectors.

6.5.2. *Typeface Conventions.* See previous section on vectors.

6.5.3. *Zero Vector.* See previous section on vectors.

6.5.4. *Negating a Vector.* See previous section on vectors.

6.5.5. *Linear Algebra.* See previous section on vectors.

6.5.6. *Vector Magnitude.* See previous section on vectors.

6.5.7. *Scalar Multiplication.* See previous section on vectors.

6.5.8. *Normalized Vectors.* See previous section on vectors.

6.5.9. *Vector Addition and Subtraction.* See previous section on vectors.

6.5.10. *Distance Formula.* See previous section on vectors.

6.5.11. *Vector Dot Product.* See previous section on vectors.

6.5.12. *Vector Cross Product.* See previous section on vectors.

6.5.13. *Linear Algebra Identities.* See previous section on vectors.

6.6. **Chapter 6.**

6.6.1. *Matrix - Mathematical Definition.* See previous section on matrices.

6.6.2. *Matrix - Geometric Definition.* See previous section on matrices.

-

A chapter that might be nice to read to become familiar with how to effectively use C++.

6.7. **Chapter 7.**

6.8. **Chapter 10.**