CS 5400

Manipulation of pictorial data by a computer

Graphics system components

CPU

GPU

Has its own memory

Memory

Framebuffer

Chunk of memory that is in the GPU

Output devices

Cathode ray tube

LCD liquid crystal display

White light shines through 2 polarizers and liquid crystal to get color

OLED

Pixels themselves emit the light (RGB)

Pixels and framebuffer

Modern graphics – raster based

An array of pixels

Entire array is called the framebuffer

Depth or precision

Bits used to represent each pixel

How many colors are represented

True color / full-color

Has 24 bits per pixel, 8 for red, 8 for green, 8 for blue

High dynamic range (HDR)

Ratio between max and min intensifies in image/display

HDR has 10 bits per color (1024 shades of each color including black)

Dynamic range of 12 bits per color is 4096:1

Human eye is estimated to have dynamic range of 10,000:1

4K (ultra hd) 2840 x 2180 = 8.2 million pixels

At 24 bits per pixel that is 25 MB

36 bits per pixel is 37 MB of memory

Coordinates

Framebuffer is drawn to an output device

Device coordinates = screen coordinates

Upper left is (0, 0) bottom right is the resolution (max x and y)

World coordinates

Device independent representation of data before it is rasterized for display

X and y between -1 and 1 and converted to device coordinates upon display

World to device

X = deviceSizex / 2 + wordX \* deviceSizeX / 2

Device to world

WorldX = (DeviceX – DeviceSizeX / 2) / (DeviceSizeX / 2)

GPU

Started as a fixed function (built-in functions)

High-degree of parallelism, thousands of vector processing units

Thousands of instructions per clock-cycle

SIMD vector and matrix operations (single instruction for matrix multiply)

Dedicated memory including the framebuffers from which the display is generated

Performs special graphics operations

PC graphics APIs

OpenGL

1992

Cross-platform

No longer under development

Only one thread issuing graphics commands

OpenGL ES (embedded systems)

2003

Cross-platform

Up to date with OpenGL

For mobile systems

WebGL

2011

Cross-platform

For doing OpenGL in browsers

Based on OpenGL ES

DirectX

1995

Microsoft’s graphics api

Vulkan

2016

Cross-platform

Successor to OpenGL

Not backwards compatible with OpenGL

Lower overhead, more fine-grained control than OpenGL

Same Api for PC and mobile

No longer a global state, per object state instead

Metal

Apple’s graphics API

Only for apple devices

WebGPU

Shading language based in Rust instead of a C language

Unit 2

Line drawing algorithms

Digital differential analyzer

How many points (units) a line takes

If ∆X > ∆Y then number = |X∆| + 1, else number = |∆Y| + 1

Bresenham

Difference of distances using a decision parameter Pk

Slope between 1 and 0

Pixels are at integer coordinates, their center is at half-integer coordinates

Pk is the current decision parameter

Pk+1 is the next decision parameter

D1 is the distance from the y-intercept at Pk+1 at yk: y – yx or m(Xk + 1) + b – yk

D2 is the distance from the y-intercept at Px+1 **+ 1**: (yk + 1) – y or yk + 1 – m(xk + 1) – b

The next pixel is xk+1 = xk + 1

Yk+1 = d1 – d2 ≥ 0? Yk + 1 else yk

…

Pk = 2∆y(xk)- 2∆x(yk) + c

c = 2∆ y+∆x(2b -1)

How we use it

If pk ≥ 0 then yk+1 = yk (+1)

If pk < 0 then yk+1 = yk

Midpoint

Two-step

Symmetric two-step

Line drawing project

Write bresenham line drawing algorithm

Make an animation with it

Function render() {

graphics.drawline(x, y, x2, y2, ‘rgb(0, 0, 255)’)

Bresenham cont

P0 = 2∆y-∆x

If pk ≥ 0 then yk+1 = yk + 1

Pk+1 = pk + 2∆y – 2∆x

If pk < 0 then yk+1 = yk

Function animationLoop(time) {

update(); //update data

render(); //sets up but doesn’t draw

requestAnimationFrame(animationLoop); //tell the browser to call the function again when it gets a chance

}

function render() {

graphics.clear();

graphics.drawLine(ptCenter.x, ptCenter

//sends commands to the browser command buffer

}

requestAnimationFrame(animationLoop);

Rasterization

Taking an image described in a vector format and converting it to a raster image

Get outline of triangle and draw horizontal lines to fill it between pixels that touch the edge of the triangle

Curves

Better approximate of natural shapes

Better describe the motion of objects

Curve design criteria

Local control of shape

Smoothness and continuity

Ability to evaluate derivatives (or directly provide slopes)

Stability (Runge phenomenon)

Ease of rendering

Parametric line form

p(u) = (1 – u)p0 +up1

smaller u = more precision

x(u) = axu­3+b …

matrix form

x(u) = [u3 u2 u 1] [ax bx cx dx] or U\*Cx

y(u) = [ “ same with y’s]

piecewise

different geometry between p0 – p1 and p1 – p2

needs points between p0 – pn

slope = dy du / dx du

runge phenomenon

higher order polynomials have more curves between two points

curve techniques

hermite curve (spline)

interpolating (curve goes through the control points), piece-wise, cubic polynomial, where the tangent is specified at each control point

find abcd for x and y

x(0) = p(0)x

x(1) = P(1)x

P(u) = MH \* Mg

Parametric form : x(u) = U\*MH\*Mgx | y(u) = U\*MH\*Mgy

cardinal spline

interpolating, piece-wise, cubic polynomial, where the slope is computed from points adjacent to the curve control points

-also known as catmull-rom spline/curve when t=0

-very similar to the hermite splines, instead of specifying slopes, computed from adjacent points

Compute Pk, Pk+1, Pk-1, Pk+2

Goes through k and k+1 but not necessarily k-1 and k+2

General form: P(u) = U\*Mspline \*Mgeometry

M for matrices

s = (1 – t)/ 2

Bezier curve

Interpolating spline (through start and end points but not intermediate)

Developed by pierre Bezier for use in designing Renault automobile bodies

Because of their properties, they are used to define the curves for fonts and used for font rendering

Can have any number of control points, but usually kept to 3 or 4

Bernstein polynomials

A Bezier curve of 4 points has degree 3

C(n, k) = n! / k!(n-k)!, only computed once

4/16

Assignment 2

Hermite, cardinal, Bezier \* 2

Start with hermite

Every equation goes once with respect to y and once for respect to x

Type

Control points: tangents

How many segments it’s broken into

Bezier

Blending function: A picture containing text, font, white, graphics

Description automatically generated

(once for x and once for y)

Pk = control point for x

U based on number of segments

N = degree of curve ( 4 points = degree 3)

K = which degree we’re working on

Cardinal spline

Doesn’t use tangents

Pk -1 not on the curve pk + 2 not on the curve

Multiply matrix once for y and once for x

Tension, how closely the curve is to the points

Hermite

Uses tangents

Line drawing review

Combine p0 into loop

Pass in x increment (-1 or 1)

Closure and memo-ization

Let factorial = function () {

Let f = [];

Return function inner(n) {

}

}();

//outer function executes immediately, inner function assigned to factorial

//previously computed values stored in f

Memo-ize C(n, k)

Let BlendC = function () {

Let memo = [];

Function compute(n, k){

Return factorial(n) / (factorial(k) \* factorial( n – k));

}

Return function inner(n, k) {

if (n > memo.length -1) {

memo[n] = [];

memo[n][k] = compute(n, k);

}else if (n < memo.length && k > memo[n].length – 1) {

Memo[n][k] = compute(n, k);

}

return memo [n][k]

}

//n will always be 3 for a cubic

Precompute with for loop up to 3

For (let k = 0; k < 3; k++) {

BlendC(3, k);

}

Let respectToU(u) {

Let memo = []

Return function inner(u) {

Let u1000 = Math.trunc(u \* 1000);

If(memo[u1000] == undefined) {

Memo[u1000] = compute(u);

}

Return memo[u1000];

}

//Use to precompute (2u^3- 3u^2 + 1) plus all ‘u’ sections

//precompute to memorize before going to render?

//code without memo-ization first then add later

//bresenham equation in quiz 2

4/18

Tension can be part of control

Affine Transformations

Translation

Moving one point by a vector giving us a new point

V(x, y) = P’ (x, y) – P(x, y)

dx = x’ – x

dy = y’ - y

Scaling

Change size

Scale each point P by a scaling factor S(x, y) resulting in P’

Each point is scaled in relation to the object’s center

S(x) > 1 moves away from center

S(x) < 1 moves towards the center

Rotation

Most complex

Rotate around a pivot point

Rotate by angle theta between two vectors originating from pivot

X = r cos(fi)

Relative to axis origin not object origin

Global coordinates

Translate object to be on axis origin, rotate, then translate back out

Local coordinates

Rotate around a local origin, then translate by world coordinate at render

Homogenous coordinates

Translation P’ = I \* P + T

H = 1, then cartisian = homogenous

Lets translation be a matrix multiplication instead of addition

Multiply rotation, translation, and scale steps in reverse order of what you want to happen to get single matrix that does all steps

P = Mc \* P (Mc translates in, rotates, then translates back out, times the points P)

Rotate about a line

P’ = T2 \* R0 \* Rx \* R-0 \* T1 \* P

0 is theta, put line intercept at origin, rotate line into x axis, rotate then undo rotate and translate.

Vector space

Euclidian space

Extends vector space by adding size and distance (magnitude of a vector)

Affine space

Extends vector space by adding a point

Preserves points, straight lines and planes

Parallel lines remain parallel

Angle and distances aren’t necessarily preserved

function drawLineQ3(x1, y1, x2, y2, color) {

var deltaY = Math.abs(y1 - y2);

var deltaX = Math.abs(x1 - x2);

var pk = (2 \* deltaX) - deltaY;

var curY = y1;

var curX = x1;

if(color === 'red'){

api.octant1 = 3

}else{

api.octant2 = 3;

}

for(let i =0; i<deltaY; i++){

//d1 - d2 = positive go up, else go down

api.drawPixel(curY, curX, color);

if(pk >= 0){

curX++;

curY++;

//pk+1

pk = pk + 2 \* deltaX - 2 \* deltaY;

}else{

curX++;

//pk+1

pk = pk + 2 \* deltaX;

}

}

}

4/23

**Computation is fast but memory is slow**

Cpu

Looks for data in cache

Cache

Not in cache – stall

Main memory

Array

cpu grabs cache line, some data around the main data if it’s in the line

Javascript

Can accidentally grab objects from array one at a time with cache stall for each object

Much slower than bringing in the whole array and then indexing

Assignment 3

Primitive – anything with a bunch of points (2 or more vertices)

Connect – if line goes from last point back to first

Local – center x, y is 0, 0

Center and verts can be local or world coordinates

Only translate primitive’s center and have other points update

Rotate in world – have to translate in and back out

Hermite not required just cardinal and Bezier

Self define what center means

Left handed coordinates system

X cross product into Y gives positive Z

Positive Z goes ‘away’ from you

Right handed coordinates system

X cross product into Y to get positive Z (goes towards you)

OpenGL – righthanded with some lefthanded

Direct x – lefthanded

Translation in 3D

T = 100t1

010t2

001t3

0001

Rotation in 3D

Rotation in x, y plane, x, z plane, y, z plane

About z-axis: x-y plane

About x-axis: y-z plane

About y axis: z-x plane

x-y plane (righthanded)

X’ = xcos(0) – ysin(0) (0 = theta)

Y’ = xsin(0) + ycos(0)

Z’ = z

y-z plane

x’ = x

y’ = ycos0 – xsin 0

z’ =

Rotation about a line 3D

Line endpoints pa – pb

1. Translate pa to origin t1
2. Rotate line into a coordinate axis (2 rotations), Ra Rb
3. Rotate by the specified ø about the coordinate axis
4. Rotate back by negative rb ra
5. Translate back by -t1

P’ = -t1\*r-a\*r-b\*r0\*rb\*ra\*t1\*P

Gimbal lock

Point gets stuck in two access planes and can’t mathematically get out

Need 4-dimensional complex number to do rotation in 3D but 2-dim complex to do 2D rotation

Because 3-dimensional complex operations are undefined

-quaternion math

Arbitrary 3D rotation

Can be done on the GPU

Float32 array

Objects in array are actual numbers and not references to numbers that need to be retrieved form memory

Modern graphics pipeline

Vertices -> vertex shader (programmable stage) -> tessellation shader (programmable) -> geometry shader (programmable) -> clipping & primitive assembly -> rasterization -> fragment shader (programmable) -> framebuffer

Programmable – has inputs and outputs that go to next stage

Shader – just a program that runs on the GPU

Vertices – can be reused between frames

Primitive types – triangles and quads (4-points)

Vertices - Triangles

1. Uniquely identify each vertex
2. Triangle strip, 2 points shared with another triangle,
   1. Each additional triangle needs 1 new point after the first
   2. Points shared by at most 2 triangles
3. Triangle fan
   1. One vertex shared by many triangles, 1 new point per additional triangle

Vertex shader

Programmable stage where each vertex is transformed from local to projected coordinates

One to one mapping of input to output, vertex is copied

Lighting and color can be computed here but is usually done later

Tessellation shader

Take patches, group of vertices and transform into new patch of vertices

Patch – group of primitives

Not 1 to 1, can increase/decrease number of primitives

Groups in – group of some size out

Geometry shader

Accepts a primitive and outputs zero or more primitives

Clipping and primitive assembly

Vertices are assembled into primitives

If not visible it is clipped out

If partially visible, subdivided into new primitives to fit screen

Rasterization

Primitive comes in and is turned into pixels

Processed into fundamental fragments (fragment data can be pixels but not always pixels)

Painter’s – draw things in the back first then things that are more in foreground on top

z-test - compares the z value of the fragment with the depth buffer

depth buffer – array index number is fragments, value is its depth

if it has less depth, override what was before

alpha blending – computationally slow

fragment shader

determine frame buffer properties for each fragment

outputs: color for each channel, depth, normal

data from previously computed framebuffers can be combined in

we’re doing vertex and fragment shaders

shading languages

shaders always run on GPU

spir-v (Vulcan) – can have other languages compile into it

HLSL(directx)

GLSL(openGL)

WGSL(webGPU)

Shaders grouped into blocks of size 32 or 64

All possible instructions are executed in parallel (all possible if statements)

Simplest shader

Vertex shader;

void main(void) //entry point

{

gl\_position = ftransform();

}

Double buffering

Two frame buffers so one isn’t being rendered while it’s being displayed

Flipped between refreshes so one is being rendered while the other is displayed

OpenGL

Low level rendering API intended for use in interactive applications

Open standard maintained by Khronos Group

OS independent, easily portable

Hardware platform independent

Language independent; exposed with C based API that other languages can call on

openGl 1 based on iris GL by SGI

openGL ES removed fixed functions

fixed function

passed in vertices, couldn’t control shading or lighting equations

openGL 4 competes with DirectX 11

openGL 4.3 adds compute shaders, fully compatible with ES 3.0

WebGL 2.0 and Vulcan succeed openGL

Mip mapping

Tell GPU to render less detail in texture as object get farther away

Off-screen rendering

render to off-screen frame buffer

instancing

reuse a 3D model instead of having multiple copies in memory

WebGl Pipeline

Javascript puts things into memory (geometry, textures)

Vertex shader accesses memory and outputs varyings into fragment shader

Fragment shader can take from memory and vertex shader and put into framebuffer

First webGL

Define html canvas element

Obtain the webGL context

Prepare the raw data (geometry)

Prepare shaders

Prepare buffer objects

Specify buffer object attributes

Request animation frame (webGL handles double buffering)

Reset framebuffer and depth buffer

Draw primitives

…

Html

<canvas id = “canvas main” width = “800” height = “600”></canvas>

Set width and height to constant values

Let canvas = document.getElementById(‘canvas-main’);

Let gl = canvas.getContext(‘webgl2’);

//obtain context one time and keep reference

Prepare raw data

Buffer objects live in GPU memory, we get references and send commands to manipulate them

Primitive is combination of 2 buffers – vertex and index buffer

Vertex buffer

Bindbuffer, set buffer commands are going to

Array\_buffer: buffer of actual data

Element\_array\_buffer: buffer of indices of data

Bindbuffer(null) unbinds buffer

Color buffer

Create buffer

Bind to buffer

Buffer the data, send it over

Unbind

Index buffer

Can be used for color and indices

5/30

Step 5 Prepare shaders – vertex shader

OpenGL 3.0 ES or higher for assignments

aPosition – floating point

in aColor – RGB plus alpha channel

vColor = aColor – input is same as output

can possibly have more than 1 input or output value

create buffer object then shader object

shader object refers to what is on the GPU

only need 1 vertex shader for multiple projects

fragment shader

we already set color at each vertex in primitive, interpolate inbetween colors

get string of shader into memory (from file)

create fragment shader object

transfer source to shader object and compile

(same steps as vertex shader)

Shader program

Create program

Attach vertex and fragment shader

Link program

Set webGL state to use this shader program

Enable position and color

Bind buffer

Get location in shaderProgram

Enable vertex attribute array on that position

vertexAttributePointer

3 – components per vertex

False – normalized to range or not

BYTES\_PER\_ELEMENT \* 3 – bytes per group of 3 elements

0 – offset for stepping through data

Set shader program in drawObject?

Reset frame and depth buffer

G1.cleardepth – z buffer / depth value

1 – further away

0 – right in front of you

In render – gl.clearColor, gl.cleardepth etc.

Gl.drawElements – just adds command to command buffer that gets executed by GPU

Canonical vs world viewing volume

Perspective vs parallel projection

Assignment 4

Get single triangle rendered first – introduction to webGL slides

First 6 – initialization

7 – 10 drawing

Array for vertices, index of each vertex, color at each vertex

Set attributes after buffers are prepared, new bind and unbind

Polygon vertex order

OpenGL expects vertices in counterclockwise winding order

Only one side of a polygon is visible

Shader files

Simple.vert:

#version 300 es

in vec4 aPosition;

in vec4 aColor;

out vec4 vColor;

Simple.frag:

#version 300 es

Promises

Let promise = new promise(function(resolve, reject) {})

Call resolve or reject based on if promise returns successfully

-passed in as parameters

Non-blocking, code execution moves on past it

.then is called when response returns – also asynchronous

Reject calls .catch

Async function

Executes and returns immediately

Inside code can wait on promises with “await”

Web server

Command line: node server.js

Requesting a file

Fetch immediately returns a promise

return fetch(filename)

.then(response => response.text())

function getFile(‘shaders/simple.vert’)

.then(source => {

let shader = gl.createShader(gl.VERTEX\_SHADER)

chain promises to load shaders from a file

declare shader variable outside of function

matrices defined in javascript

shader takes matrices and multiplies vertices to make transformations

model matrix

from local to world

apply rotation in local coordinates than translate to world

view

location of camera

projection

frustum, near, far, left, right

fiven to GPU

all three of these matrices multiplied against each vertex

order is model – view – projection

in matrix mult = projection \* view \* model \* vertex

shader uniforms

value is the same for all vertices or fragments

same across the draw call

matrices specified in here

uniform mat4 uThing

3 different uniforms, one for each matrix

All faces made of triangles in counter-clockwise order

Framework code now has node server and matrix multiply

Declare 3 uniforms for each matrix

Each shader has 3 matrices

Perspective

COP = center of projection

Closer COP is to the near clipping plane, the wider the field of view is

Near and far planes set in world coordinates

Triangle in world coordinates – (0,0) is the center -1 and 1 for top and bottom

View matrix – negative Z moves camera forward

Translation – last column of matrix does the translation

Matrix Order: Local to world then camera then projection

Perspective looks smaller in the distance

Put near clipping plane at 1, far at 10

GLSL shader language

Int, float, double, uint, bool

Containers:

Vectors – mathematical not an array, 2, 3 or 4 components

matrices – floating point matrix of values

vectors

vec[n]: vector of ‘n’ floats, bools, etc

access values - vec.x, vec.y, vec.z, vec.w

constructors

vec3 myVec = vec3(0.1, 0.2, 0.3)

swizzling

myVec = yourVec.zyx, change order of x, y, z

matrices

NxN with Ns between 1 and 4

myMat[1][2] 2nd column 3rd value

transpose, determinant, inverse (1 line and code but can be many instructions)

arrays

only 1 dimensional allowed

float myArray[4]

structs

no class types just structs

is a value itself

pass by value not reference

can be put in arrays

iteration

for, while, do while, if, if else, switch

functions

no recursion

parameters – by value

return types – by value

Triangle normals

Normal – vector that is perpendicular to the surface of the triangle

Plane – Ax + By + Cz + D

Cramer’s rule – get coefficient of A, B, and C

Cross product – make two sides of triangle vectors then use cross product to get vector perpendicular to both of them

Find sin of angle between vectors

Vertex normals

Average of the triangle normal at each vertex

Normals

Dot product of normal and vector in a plane is 0

Put matrix and matrix inverse in middle of dot product

M is the model-view matrix

Right hand becomes eye space

Left hand also in eye space

Left hand becomes transformed normal we can use in our calculations

Backface removal

Surface that faces away from the viewer

Expect 50% of polygons to be backfacing

Compare vector of viewing angle to normal coming off polygon to see if it’s backfacing

Angle between -90 and 90 it is visible

Dot product of two vectors is the cosine of the angle

Will be between -90 and 90 if cos() ≥ 0

Take normal in projected space not world space

Back culling done in clipping and primitive assembly stage so fragment shaders for

non-visible polygons aren’t computed

PLY files

Uint16 array can only index 64,000 values

Can be ascii or binary format

Based on wavefront obj format

ply

format ascii 1.0

comment [insert comment here]

element vertex 14 (14 vertices)

property float x (how many properties each vertex will have)

…

element face 25

property list uint8 (data type of how many vertices in primitive) int32 …

end\_header

faces with more than 3 vertices need to be split into triangles

face with n vertices will have n-2 triangles (triangle fan)

Lighting model

Ambient

Diffuse

Same intensity regardless of direction

Specular

Highlights that are directional

Red, green, blue channel

Define how much red blue and green for ambient, diffuse and specular channels

Specular has a shininess factor (power function)

Ambient equation: Ia = Ka\*La ( I = intensity )

Amount of light reflected is proportional to the cosine of angle between light and normal vector

Intensity diffuse = reflectivity \* light \* dot of N and L

Once for red green and blue

Illumination model

Specular reflection

Needs to be unit vectors

Projection L onto N is L dot N

R + L = (2N \* L)N

Shading model

Gouraud shading

Linearly interpolate intensity between points

Fragment shader doesn’t change

Assignment 5

Per vertex shading

Total reflection – once for red, blue and green

Ns power affects shininess

2 models, 3 different lights in the scene

Each light a different color, turned off and on

No modifications to fragment shader

Models aren’t in unit space

Cant compute vertex normal in vertex shader

Use vertex normal

Average out triangles sharing one vertex

Don’t do linear search

As triangles are read in, build array that maps indexes together (adjacency matrix )

No color buffer

Pass vertices and normal into shader

Normal are normalized (divide by magnitude)

Let adjacency = new Uint32Array[size-verts]

Adjacency[0].push([]) (push empty list for each vertex, then add adjacents later)

Use javascript .split(``) //split on spaces

.indexOf(‘something’) //use for element face and element vertex (after split lines?)

Precompute normal for every vertex

Uniforms

Matrices: Model, view, projection,

Lighting: material properties for each object, diffuse emission for each light, light position (one for red, green and blue) //pass in vec3 to pass all 3 in one vector

Lighting computed after model translated and rotated but before projection

All lighting computations need to be unit vectors – intensity needs to be 1

Use clamp function in glsl to keep intensity at 1

Generalized camera model

Camera looking vector has perpendicular up vector and sideways vector (right vector)

U – right vector

N – vector from eye to viewing target

V – up vector

Move camera to line up up right and eye with x y z

Transform world in opposite direction to change view

U into x

V into y

N into z