Computer Graphics

Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison CS559

Teacher: Prof. Michael Gleicher

Figures included in this note are from Prof. Gleicher's slides unless otherwise stated.

Introduction

In general, **computer graphics** (CG) is the study of how computers create visual things we see. CS559 is, in particular, about **how to program computers to draw & animate**.

It is NOT about answering these questions:

- What pictures to make
- How to use the tools & APIs
- Any specific application

CS559 uses the *web programming* set of APIs (with JavaScript ES6+) to introduce computer graphics. Check out the MDN web doc for references. There have several advantages for beginners:

- Modern browsers are powerful and provide a full set of JS environment out-of-the-box
- They are portable, uniform, and less machine-dependent
- Web programming is relatively fun & satisfying to learn
- There are interesting CG frameworks for modern JS, even having direct hardware access support

Remember that, no matter how the APIs change, the core ideas of CG remain the same. High-performant game engines will prefer lower-level C++ frameworks with various optimization techniques, yet JS frameworks are better for beginners to play with.

Basic Ideas in Graphics

Modern displays typically use the **RGB value** (tuple of three values from 0~255, representing the concentration of red/green/blue) to compose a visible color on the wavelength spectrum. Sometimes it comes with an A (alpha) component which stands for the *opacity*.

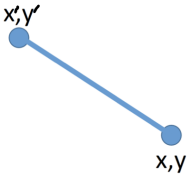
Photons travel in straight lines. The way our eyes work leads to the fact that we can be easily faked out on *depth* and *distance*. To be more precise, **we sense 2D** and **we infer 3D**.

There are two ways to create images that describe what we see:

- *Physically-based*: simulate photons
- *Primitive-based*: simulate painting

Most computer graphics use the painting approach. In this approach, there are two ways of representing (both 2D and 3D) images:

- **Geometric** (*Primitives*; *Objects*, 对象): represent lines & shapes in objects, described mainly by coordinates & *vectors*



- **Sampled** (采样; *Raster*, 栅格): use a *grid* of color points/*pixels*

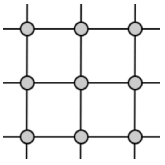


Image representation needs to be carried out on physical media, called **displays**, to let us see. Accordingly, there are two types of display devices:

- Geometric (Primitives): pen plotters, laser light shows, CRT vector scanners, ...
- Sampled (Raster): LCD, LED, laser/inkjet printers, most 3D printers, projectors, film, ...

Most modern displays are sampled, even if programmers draw in a vectorized language since it is more human-understandable. The automated process of converting a geometric image into a sampled image ready for display is called **rendering** (渲染).

In displays, several techniques are used to let us see more comfortably. Displays have *buffers* (frame buffer, color buffer, ...) to hold the data describing the image. They *flicker*/*strobe*, i.e. redraw at a frequency high enough, e.g. 30 Hz, to cheat our eyes to believe that we are seeing a continuous movement. This is called the **flicker fusion**. The shown image for a given interval is called a **frame** (帧) and the flickering rate is called the *frame rate* (帧率).

A flicker-based display need to satisfy the following two properties to make us feel comfortable:

1. The frame rate needs to be high enough, and,
2. The frame rate needs to be consistent: our eyes can sense changes if the frame rate is not stable.

> Another term, *persistence of vision*, describes a different thing that a single flash of an image may last in our mind for more time than it actually stayed.

Display synchronization (*Buffering*) describes the action of a computer graphics hardware putting data into a buffer and showing the buffer at proper times. We often do **double buffering** (双重缓冲), where we have two buffers and we alternate between them. It has several advantages:

1. We clear & draw on buf2 behind the scene when the system is showing buf1. It helps with frame rate constancy, as long as buf2 has done processing before the start of the next frame.
2. Even if we couldn't finish drawing buf2 in time, the system can choose to just redisplay buf1 for the next frame, without showing a partially corrupted image.

> Nowadays, we even have triple or higher buffering.

Web Browser APIs

Web browsers have several kinds of graphics APIs:

- Canvas: an HTML-5 built-in 2D canvas API
  - It is *immediate*: draws whatever we tell it to and does not keep a reference to what it has done
- SVG (scalable vector graphics)
  - Keeps a *display list* of graphic objects we created, could be stored as .svg files
  - Objects are DOM elements, so can be manipulated with JS
- WebGL: a JS version of OpenGL ES
  - Has direct access to graphics hardware
  - Gives programmers more low-level control, however we must learn to program the hardware

Some higher-layer libraries are built upon these APIs:

- Three.js: a display list API built on top of WebGL
- D3: a tool to help manipulate DOM elements, very useful for SVG

2D Transformations

One essential technique in CG is **transformation**. We draw objects according the the local **coordinate system**, and when we want to move the object around without changing the drawing code and without breaking its shape, we just move the coordinate system.

Common 2D Transformations

There are several basic 2D transformation operations. Note that they are all centered at the origin point of the current coordinate system and are applied to the coordinate system (not the objects):

1. *Translation*: shifting the coordinate system - `translate(tx, ty) := f(x, y) => (x+tx, y+ty)`
2. *Scaling*: stretching/shrinking the coordinate system - `scale(sx, sy) := f(x, y) => (sx*x, sy*y)`
   - If `sx == sy`, we call it a *uniform* scaling; otherwise, it is non-uniform
   - To scale about an object at its center (40, 40), the snippet:

     ```
     context.translate(40, 40);
     context.scale(sc, sc);
     context.translate(-40, -40);
     context.fillRect(...);
     ```

   - Using a negative scaling factor can result in *flipping*
3. *Rotation*: spinning the object around the center of the current coordinate system
   - Rotation & Translation are called *rigid* transformations as they do not scale
   - To rotate an object around its center (40, 40), the snippet:

     ```
     context.translate(40, 40);
     context.rotate(spin);
     context.translate(-40, -40);
     context.fillRect(...);
     ```

   - The "*handedness*" of a coordinate system is defined by whether going from positive x-axis to positive y-axis is *clockwise* (*right-handed*) or *counter-clockwise* (*left-handed*)

    - Point thumb into the screen, see which way the other fingers go
    - Rotation does not change handedness
    - *Reflection* (scaling where only one scale factor is negative) changes handedness
4. *Shear* (*Skew*): incline the shape - `shear(kx, ky) := f(x, y) => (x+kx*y, y+ky*x)`

Notes:

- Transformations affect drawings afterwards, but do not affect things that have been drawn
- Transformations take effect relative to the latest coordinate system, so multiple transformations will stack up if we apply them repeatedly to the same context

Transformation operations take effect sequentially. This procedure can be thought of as mathematical *function compositions*: $f(g(x,y)) = f \circ g(x,y)$. But be aware of the order here: earliest transformations on the coordinate system will affect all the following transformation on the changed coordinate system - hence, the earliest transformation in code appears as the outer-most function. Example:

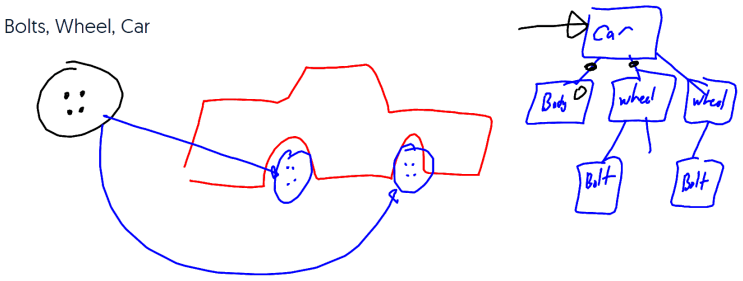
```
context.scale(3,2);
context.translate(40, 30);
context.fillRect(...);
// This maps to s(t(x, y)) => (3*(x+40), 2*(y+30))
```

To avoid transformations to pollute future drawings, it is a good practice to save & restore the context:

```
context.save();
context.translate(20, 30);
context.fillRect(...);
context.restore();
```

Hierarchical Modeling

To draw a complex picture, we often represent objects as a **scene graph**, i.e., a direct acyclic graph (DAG) or tree of objects. Every object lives in its own local coordinate system, which in turn is computed relative to its parent's coordinate system. They form rooted *articulated chains*.



We save & restore contexts as a stack. When drawing object A, the stack should consist of A's context at stack top followed by parents' contexts until the root. Since transformations of coordinate systems are often represented as matrices, this stack of contexts is often called a *matrix stack*.

Transformations as Matrices

We represent a point as a tuple of coordinate numbers. For example, a 2D point $\mathbf{x} = (x, y) = \begin{bmatrix} x \\ y \end{bmatrix}$.

Scaling, rotation, and shearing transformations are all **linear transformations**. They can be represented as a matrix:

$$
\begin{aligned}
x' &= a_{xx}x + a_{xy}y \\
y' &= a_{yx}x + a_{yy}y \\
\mathbf{x}' &= \begin{bmatrix} a_{xx} & a_{xy} \\ a_{yx} & a_{yy} \end{bmatrix} \mathbf{x} = \mathbf{A}\mathbf{x}
\end{aligned}
$$

Linear transformations have the following nice properties:

- Zero is preserved - vector zero turns out as zero, origin turns out as origin
- Composition of transformations is composition of matrices: $f(g(\mathbf{x})) \equiv \mathbf{F}\mathbf{G}\mathbf{x}$
- Matrix multiplication associates, you can multiply the matrices first: $\mathbf{A}(\mathbf{B}\mathbf{x}) = (\mathbf{A}\mathbf{B})\mathbf{x}$

- The set of linear transformations is *closed* - any sequence of linear transformations corresponds to some single linear transformation that has the same result
- Matrix multiplication does NOT commute, so the order of application matters

You may have noticed that translations are not linear transformations because they at least move the origin. The combination of a linear transformation with a translation forms an **affine transformation** (仿射变换). Affine transformations are what we will mostly focus on in this course.

$$\mathbf{x}' = \begin{bmatrix} a_{xx} & a_{xy} \\ a_{yx} & a_{yy} \end{bmatrix} \mathbf{x} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \mathbf{A}\mathbf{x} + \mathbf{t}$$

Homogeneous Coordinates

Several transformations we often use in 2D, such as translations , are not linear and are tedious to write in math. To use linear transformations to represent affine transformations, we use the trick that *embeds* 2D points into the 3D space $\rightarrow$ do linear transformations in 3D space $\rightarrow$ interpret the points back to 2D. This trick of using an $n+1$ dimensional space to represent n dimensional points is known as *homogeneous coordinates* (齐次坐标).

We will focus on the basic version where we just fix the third dimension w as 1:

$$\mathbf{x}' = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a_{xx} & a_{xy} & t_x \\ a_{yx} & a_{yy} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbf{A}\mathbf{x}$$

> Homogenous space treats all points along lines through the origin as equivalent. Our 2D space is the plane $w = 1$ of the 3D space. A point (x, y, w) in 3D is equivalent to the intersection - the point $(\frac{x}{w}, \frac{y}{w}, 1)$, hence the point $(\frac{x}{w}, \frac{y}{w})$ in our 2D space.

To read the matrix $\mathbf{A}$ above, a convenient way is:

- The 1st column (a_{xx}, a_{yx}) tells us the new direction and scale of the new x-axis
- The 2nd column (a_{xy}, a_{yy}) tells us the new direction and scale of the new y-axis
- The 3rd column (t_x, t_y) tells us the new position of the origin
- The last row is always $(0, 0, 1)$ for affine transformations

Matrix for Common Transformations

For common transformations:

Transformation	Code	Matrix
Translation	`translate(tx, ty)`	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$
Scaling	`scale(sx, sy)`	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$
Shearing	`shear(kx, ky)`	$\begin{bmatrix} 1 & k_x & 0 \\ k_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
Rotation	`rotate(r)` clockwise where y+ goes down	$\begin{bmatrix} \cos r & -\sin r & 0 \\ \sin r & \cos r & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Some notes on rotation:

- Rotation transformation preserves: *distance*, *angle*, & *handedness*
- Hence, the matrix must be:
  - Each row/col being *unit length*: $\sqrt{a_{xx}^2 + a_{yx}^2} = \sqrt{a_{xy}^2 + a_{yy}^2} = \sqrt{a_{xx}^2 + a_{xy}^2} = \sqrt{a_{yx}^2 + a_{yy}^2} = 1$
  - Rows/cols are *orthogonal*: dot product $a_{xx}a_{xy} + a_{yx}a_{yy} = 0$

- The *determinant* $a_{xx}a_{yy} - a_{xy}a_{yx} > 0$
- Its inverse is the transpose
- Cannot preserve *linear interpolation*

The current transformation context is a matrix - the result of a multiplication of transformation matrices. The following code:

```
context.scale(3,2);
context.translate(1, 4);
context.rotate(Math.PI/6);
context.fillRect(...);
```

, corresponds to the following transformation context for the drawing on line 4:

$$\mathbf{T} = \begin{bmatrix} \frac{3\sqrt{3}}{2} & -\frac{3}{2} & 3 \\ 1 & \sqrt{3} & 8 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{I} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

You can think about this composition in two ways, forward or backward:

1. The point goes from its local coordinate to the global (canvas) coordinate by left-multiplying matrices from right to left;
2. In actual program execution, it's the coordinate system that composes the transformation context by right-multiplying matrices from left to right. (Originally the context is the *identity matrix* $\mathbf{I}$.)

2D Curves

We will go over the formal definition and the practical solutions for drawing curves.

Parameterization

There are two ways to mathematically define a curve:

- *Implicit*: use a "test function" $f(x, y)$ that takes a point and produces a number. If the result is zero, it's on the curve
  - Example: a circle $(x-1)^2 + (y-3)^2 - 4 = 0$
  - This definition is handy in doing mathematics, but hard to compute for computer programs
- *Parameterized* (参数化): take a "free parameter" $t \in$ some range. Define $f(t)$, or more explicitly $f_x(t), f_y(t)$ for 2D, that takes in the parameter and produces a point position
  - Think of the free parameter t as the "time" of drawing, and the result as the current position of the tip of the pen on a 2D paper
  - Example: a circle $f_x(t) = 1 + 2\cos(2\pi t)$, $f_y(t) = 3 + 2\sin(2\pi t)$, $t \in [0, 1]$
  - We often use u to denote a unit free parameter of range $[0, 1]$, WLOG

We use *curve* to define the final shape on paper, and use **parameterization** to define the mapping from free parameter to pen tip position. Note that different parameterizations may produce the same curve: you can have your pen move at different "speed"/"direction" at different times but still draw the same shape.

Take a line segment from $\mathbf{p_1}$ to $\mathbf{p_2}$ as an example, the parameterization $f(u) = (1-u)\,\mathbf{p_1} + u\,\mathbf{p_2}$ maps to the curve.

Connectivity, Continuity, & Tangents

A curve can be *disconnected* but still being one curve. It's just the pen jumps at the point of disconnection.

Connectivity is actually a special case of **continuity**. We say a curve is *continuous* (to be more precise, $C(0)$-continuous) at a parameter value t if $f(t^-) = f(t^+)$. In other words, the curve is continuous here in its zeroth derivative. More generally, if the curve is continuous in its i-th derivative, we say that the curve has $C(i)$ continuity.

- $C(0)$ continuity means the curve is connected
- $C(1)$ means the curve has no "corners", i.e., $f'(t)$ is continuous; A V-shape is $C(0)$-continuous but not $C(1)$
- $C(2)$ means the curve looks very smooth, i.e., $f''(t)$ is continuous; Two connected quarter-circles switching direction is $C(1)$-continuous but not $C(2)$

Note that a pen may draw in different speeds but still producing the same curve. In the above definition of continuity, if a pen draws a line segment slowly at the beginning but suddenly speeds up, we will say the line segment is not $C(1)$. However, we generally do not care about that speed change. We denote **geometric continuity** as $G(i)$ - the i-th derivative of the parameterization have no discontinuity in their *direction*. This ignores the "scale" of the derivative.

Polynomial Pieces

A complex curve can be made from multiple small *polynomial* pieces. A polynomial of degree d:

$$f(t) = \mathbf{a_0} + \mathbf{a_1}t + \mathbf{a_2}t^2 + \cdots + \mathbf{a_d}t^d$$

In 2D, each of these $\mathbf{a_i}$ is a 2D vector. We say the the curve *interpolates* $\mathbf{a_0}$ at the *site* $t = 0,$ etc.

In CG, we often use *cubic* polynomial pieces, i.e., of degree 3.

- If we only use linear segments, we can at best have $C(0)$; quadratic polynomials are not good enough
- Higher degree polynomials are typically not necessary and are harder to compute

We compose the parameterization of a complex shape by defining the function on multiple cases of unit ranges:

$$f_*(t) = \begin{cases} f_0(t-0), & \text{if } 0 \le t < 1 \\ f_1(t-1), & \text{if } 1 \le t < 2 \\ \dots & \\ f_m(t-m+1), & \text{if } m-1 \le t \le m \end{cases}$$

Now, if we want $C(0)$ continuity, i.e., connectivity, we make sure that every $f_i(1) = f_{i+1}(0)$. If we want $C(1)$ continuity, we make sure that every $f_i'(1) = f_{i+1}'(0)$, etc.

Hermite Forms

A cubic polynomial has a *degree of freedom* of 4, as it has 4 tunable coefficients; We often use the positions and derivatives of the beginning and the end point to specify a certain cubic segment - this is called the **Hermite form** of specifying polynomial segments (more precisely, a 3rd degree Hermite form curve).

Given the four *control points* $\mathbf{p_0}, \mathbf{p_0'}, \mathbf{p_1}, \mathbf{p_1'}$, we can derive the four coefficients:

1. $\mathbf{a_0} = \mathbf{p_0}$
2. $\mathbf{a_1} = \mathbf{p_0'}$
3. $\mathbf{a_2} = -3\mathbf{p_0} - 2\mathbf{p_0'} + 3\mathbf{p_1} - \mathbf{p_1'}$
4. $\mathbf{a_3} = 2\mathbf{p_0} + \mathbf{p_0'} - 2\mathbf{p_1} + \mathbf{p_1'}$

In matrix form:

$$f(u) = \begin{bmatrix} u^0 & u^1 & u^2 & u^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & 3 & -1 \\ 2 & 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p}_0 \\ \mathbf{p}_0' \\ \mathbf{p}_1 \\ \mathbf{p}_1' \end{bmatrix}$$

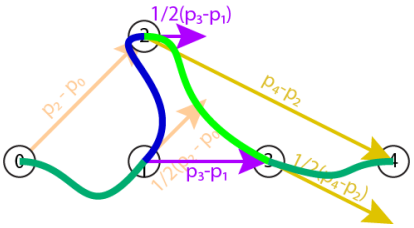
We could multiply the u's with the middle matrix first. This gives us a notation in *basis functions*:

$$\begin{aligned} f(u) = b_0(u)\,\mathbf{p}_0 + b_1(u)\,\mathbf{p}_0' + b_2(u)\,\mathbf{p}_1 + b_3(u)\,\mathbf{p}_1' = (1 - 3u^2 + 2u^3)\,\mathbf{p}_0 + \\ (u - 2u^2 + u^3)\,\mathbf{p}_0' + \\ (3u^2 - 2u^3)\,\mathbf{p}_1 + \\ (-u^2 + u^3)\,\mathbf{p}_1' \end{aligned}$$

An important reason why we choose Hermite forms to do the interpolation, instead of using many other options (e.g., specifying 4 derivatives at the beginning, etc.), is that Hermite forms let us use the position and 1st-derivatives at the beginning and the end, so that we can easily achieve $C(1)$ continuity when connecting cubic pieces together.

Cardinal Splines

To actually connect multiple pieces of cubic polynomials together to form a complex shape, we often use a interpolation technique called *cardinal spline*.



We first specify a sequence of intermediate points. We then draw a cubic segment between each pair, where the derivate at a point (used as a beginning or as an end) is computed as:

$$\mathbf{p}'_\mathbf{i} = s\Big(\mathbf{p_{i+1}} - \mathbf{p_{i-1}}\Big)$$

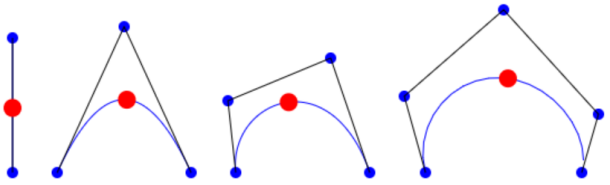
, where $s = \frac{1-\tau}{2}$ is often chosen as $\frac{1}{2}$, i.e., τ, the *tension*, is 0. If so, this is called a "Catmull-Rom" spline.

Bézier Curves

A problem with cardinal spline is that we have to interpolate through all the intermediate points. We often do not want to go through all the points, to make the result look "smoother". Non-interpolating curves are called *approximating* curves. One such technique is the *Bezier curve*.

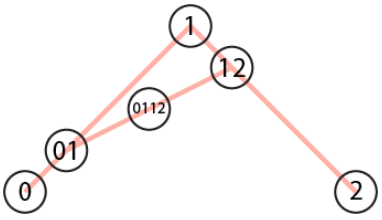
- Bezier curve interpolate their first & last points $\mathbf{p_0}, \mathbf{p_n}$
  - We add arbitrary number of $n - 1 \geq 0$ control points to influence what happens between them
  - Bezier curves are polynomials of degree n
  - Bezier curves do not have *locality* - the curve depends on all of its control points
- Bezier curve stays within the "convex hull" of control points
  - The tangent at the beginning point $\mathbf{p_0}' = n(\mathbf{p_1} - \mathbf{p_0})$
  - Then, the curve does not make more than $n - 1$ changes in its direction
- Bezier curves are *symmetric*: going forward gives the same result as going backward
- Bezier curves are *variation diminishing*: if you draw any straight line, the curve crosses that line at most n times
- Bezier curves are *affine invariant*: performing an affine transformation on the control points of a Bezier curve gives a Bezier curve

Examples:



Bezier curves are constructed through a recursive process called the *DeCastlejau Construction*:

1. Between any consecutive pair of points, linearly interpolate u along the way - this results in $n - 1$ points
2. For the new points, if there is more than 1 point left, repeat the process with the new set of points
3. Repeat until we have one final point - its trajectory with u going from 0 to 1 is the Bezier curve

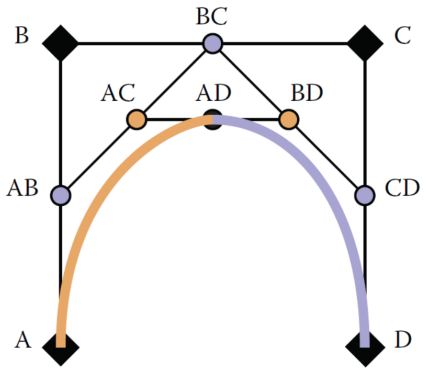


By writing out the interpolations, they form a nice set of basis functions as well. In practice, we often just use the $n - 1$-th degree basis functions to compute Bezier curves efficiently. This approach is called the *Bernstein basis polynomials*.

$$\mathbf{f}(u) = \sum_{i=0}^{d} b_{i,d}(u)\ \mathbf{p_i},\ \text{where:}$$

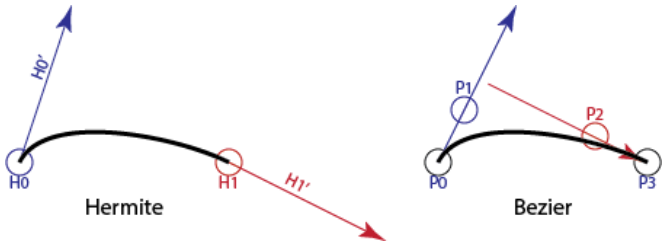
$$b_{k,n}(u) = C(n,k)\ u^k\ (1-u)^{(n-k)} = \frac{n!}{k!\ (n-k)!}\ u^k\ (1-u)^{(n-k)}$$

A Bezier curve can be split up at the point of u by drawing out the DeCastlejau construction at u:



This gives a split of curve $[A, B, C, D]$ into two curves $[A, AB, AC, AD]$ and $[AD, BD, CD, D]$.

Cubic Bezier curves are the most commonly used ones in CG. For them, we can have a similar API of "control points" like Hermite forms for cubic polynomials:



, where the math is:

$$
\begin{aligned}
\mathbf{p_0} &= \mathbf{h_0} \\
\mathbf{p_1} &= \mathbf{h_0} + \frac{1}{3}\mathbf{h'_0} \\
\mathbf{p_2} &= \mathbf{h_1} - \frac{1}{3}\mathbf{h'_1} \\
\mathbf{p_3} &= \mathbf{h_1}
\end{aligned}
$$

Bezier curves have the following disadvantages, where we may prefer other options:

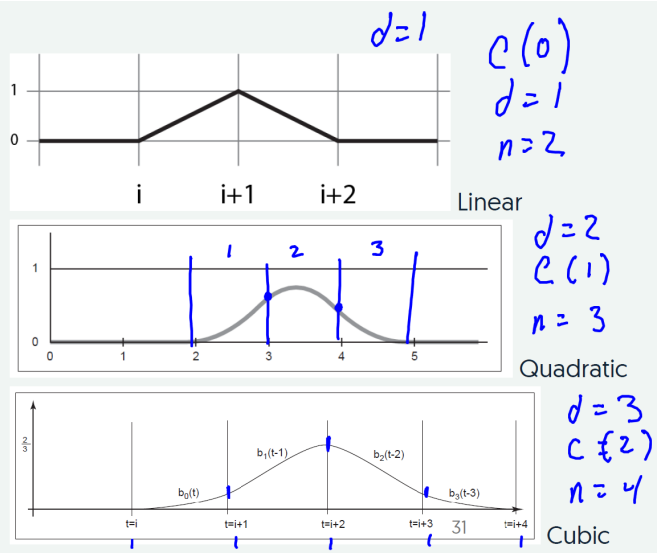
- For getting $C(2)$ and higher continuity and making the curve really smooth, we way want to use *B-Splines*
- Beziers are *affine invariant* but not *projective invariant*: dividing it by another polynomial (the projection target curve/plane) does not guarantee to produce a polynomial; We may want to use *rational curves*

B-Splines

A geometric *subdivision* approach to B-splines is the *Chakin corner cutting* algorithm:

1. Take the control points, connect together
2. Subdivide each line segment at $\frac{1}{4}$ and $\frac{3}{4}$, then cut the corners, only connect all the middle pieces
3. Go indefinitely to the limit, converges to B-spline

Algebraically, *uniform* B-spline basis functions are:



- In a B-spline of degree d, a control point is *active* in at most $d+1$ *phases*; This means at any given parameter value, $d+1$ control points are active, each in a different phase
  - The whole curve meets $C(d-1)$ continuity
  - B-spline of degree 1 is just a linear chain of control points
- The curve is *non-interpolating* for $d > 1$ - it doesn't go through the control points

Arc-Length Parameterization

If we simply connect pieces of different lengths together where each piece has a parameter u going from 0 to 1, we would have the pen moving at different speeds when drawing the whole shape. This can also happen on the parameterization for a curve, e.g., $f(u) = (u, u^2)$ for a parabola. This is called a *non-arc-length* parameterization.

In contrast, if the pen moves at a constant speed, it is called an *arc-length* parameterization. This is useful when we want to animate something that moves along a complex shape at constant speed. We could map a non-arc-length parameterization to an arc-length parameterization by *sampling*:

- Make a table of several samples of $u \to$ total distance along the curve, e.g., $0 \to 0.34; 0.1 \to 0.79; \dots$
  - The finer-grained sampling, the better speed constancy
  - Distance is hard to get analytically for high-degree curves; Typically we break into small linear segments per parameter value progress and add up the lengths of the linear segments
- We progress the distance at constant speed, use the distance to reverse-lookup which u region we are in and approximately which u value we should be at

3D Transformations

We assume using a scene-graph API like THREE.js when working with 3D graphics.

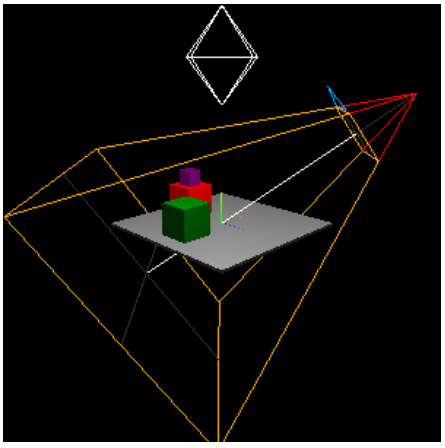
3D Drawing Elements

To draw a 3D picture, these are the basic elements we need:

- A 3D **world**, i.e., the **scene**: the scene graph, which is a tree of objects
- A **renderer**: the space we draw into, also refers to the procedure of converting a 3D scene-graph (object tree) into a proper 2D picture
- A **camera**: the *viewing transformation* from the 3D *world* to the 2D canvas we draw into, think of it as a *projection* from the 3D world into a 2D camera lens (with something more)
  - Camera is typically also considered a "zero-volume" object in the world, looking at its negative z-axis
  - Its *field of view* (FOV) defines how wide-angle the lens is
  - Anything too close to its near distance or too far away from its far distance will be ignored
- Every object is:
  - Has its *geometry*: a collection of triangles, composing the shape of the object
  - Has its *material*: the appearance of the object, color, and how it interacts with lighting; can also have global effects which takes other objects into consideration, e.g., shadows

    - THREE calls the combination of an object's geometry and material as a *mesh*, while in theory, mesh refers to only the geometry
- Some **lighting**, otherwise the image would be completely dark
    - *Ambient* lighting (环境光): unreal, uniform lighting coming from all directions
    - *Point* lighting (and *spotlight*): lighting that has its source and goes to a cone of directions; typically also considered "zero-volume" objects in the world
    - *Directional* lighting: parallel rays of light like the sunlight; considered "zero-thickness" planes in the world

An example of a perspective camera on a world with a few objects and a point lighting:



Barycentric Coordinates

A barycentric coordinate is defined by a simplex. Say we have a triangle of points (v_1, v_2, v_3), then a point p can be represented as:

$$p = \lambda_1 v_1 + \lambda_2 v_2 + \lambda_3 v_3, \text{ where } \lambda_1 + \lambda_2 + \lambda_3 = 1.$$

We call $(\lambda_1, \lambda_2, \lambda_3)$ the Barycentric coordinate of p in the coordinate system of this triangle.

- p is on the plane of this triangle
- If all λ's are positive, then p is inside this triangle

With a simplex of d vertices, the Barycentric coordinate has d numbers.

> Vectors *normal* is the cross product of vectors:
>
> - It points to the direction determined by the right-hand rule, so order matters in the cross product
> - For a triangle ABC, its normal direction can be specified as right-hand rule A-B-C, where does the thumb point to
> - Mathematically, it is $\overrightarrow{AB} \times \overrightarrow{AC}$

3D Rotations

3D rotations obey the same rules & properties as in 2D.

Notice the different between rotation and orientation:

- *Rotation* is a relative transformation, an action, think of as a translation vector
- *Orientation* is the current direction in world axis the object is now facing at, think of as the current position

The **Euler's Theorem** states that any rotation can be represent as a single rotation about some axis; Any rotation in 3D can be represented as a sequence of three rotations about some fixed axis. Say $\mathbf{p}' = R_x R_y R_z \mathbf{p}$ - we denote this as **Euler Angle** (XYZ).

- Order matters: (XYZ) $\neq$ (ZYX) $\neq$ (ZXY)
    - The one closest to object (right-most) is purely around object's *local axis*
    - The one closest to world (left-most) is purely around *world axis*
    - Say in (XYZ), if Z is zero, then Y is around object y-axis, but if Z is non-zero, Y is not around object y-axis; if X is zero, then Y is around world axis, but if X is non-zero, Y is not around world axis
    - In other words, rotation matrices around different axes are NOT commutable:

$$R_z(a) \circ R_z(b) = R_z(a+b), \text{ on the same axis, then addable}$$
$$R_x(a) \circ R_y(b) \circ R_x(c) \neq R_x(a+c) \circ R_y(b), \text{ has different axis in between, then not addable}$$

- Not necessarily three orthogonal axes XYZ; can even repeat, e.g., ZXZ

In THREE.js, the `.rotation` field of an object is an Euler Angle in the order XYZ, with the world axis being its parent's coordinate system (due to the hierarchical modeling tree):

```
obj.rotation.set(Rx, Ry, Rz);  // Object rotation transformation R = Rx Ry Rz

// API #1: directly setting value will directly modify the matrix in place
obj.rotation.y = Ry2;   // R = Rx Ry2 Rz
                  |_______________^

// API #2: relative method adds on to the current state
//         it takes in how much you wanna rotate around the parent coordinate system
//         and THREE automatically updates the Euler Angles for you
obj.rotateX(Rx2);       // R = Rx' Ry' Rz'  -- computed automatically
```

The Euler Angle approach has several problems:

- *Gimbal lock*: say in XYZ, if Y = 90 degrees clockwise, then it always aligns object-local z-axis with the world x-axis, which means that then no matter what X & Z are, the object y-axis always stays on the world $x = 0$ plane
  - Tweaking X & Z can only give us rotations around the world x-axis (which is now also the object-local z-axis)
  - We lost a *degree of freedom* (d.o.f.)!
- Cannot linearly interpolate: two different sets of Euler angles may produce the same rotation transformation (e.g., X 90 then Y 90 == Y 90 then Z 90), but any linear interpolation of the two does not!

A different approach which might be more intuitive is called the **Axis Angle**. It represents rotation as a single rotation around some world axis of some angle: $(\theta, \mathbf{v})$, where θ is the angle and $\mathbf{v}$ is the axis.

- Simple progression on an Axis Angle does NOT map to simple progression on the Euler Angle representation
- Problem: Hard to compose two Axis Angles with different axes

Computation on the Axis Angle representation is typically done through conversion to **quaternions**: $(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\mathbf{v})$.

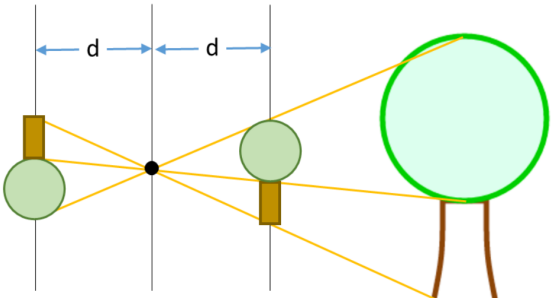
- This is a *unit* quaternion: has magnitude of 1 (given that $\mathbf{v}$ is already a unit vector)
- A bunch of good properties: can be multiplied and composed, can do *spherical linear interpolation* ("*slerp*")
- THREE stores Axis Angles internally as quaternions
- Euler Angle (XYZ) can be converted to quaternion by making a quaternion for each and multiply together

Projections

There are three types of *projection transformations* (投影):

- *Orthographic*: not realistic
- *Isometric*: not realistic
- *Perspective* (透视): realistic

Perspective projection camera is basically:



- The black point is called the *focal point* (焦点)
- The length d is called the *focal length* (焦距): $\mathbf{p}_{film} = \frac{d}{\text{actual distance z}}\mathbf{p}_{world}$
- Often we clip things that are too close or too far. We call the distance range (from focal) in which we capture world points the *frustum* (视锥) = $[n, f]$; it gets mapped to $z = [-1, 1]$ in computation
- If the film is put behind our focal point, then it is often called a *pinhole* camera

In homogeneous coordinate matrix, the projection matrix looks like:

$$\mathbf{Px} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Overall, the transformation matrices that will be applied to an object in tree looks like:

$$\mathbf{X}_{screen} = (\mathbf{P}\ \mathbf{C}^{-1})\ (\mathbf{T}_{parents}\mathbf{R}_{parents}\mathbf{S}_{parents})\ (\mathbf{T}_{obj}\mathbf{R}_{obj}\mathbf{S}_{obj})\ \mathbf{x}_{obj}$$

- First part is the *viewing* transformation, composed of:
  - $\mathbf{P}$ is the projection matrix
  - $\mathbf{C}$ is the camera transformation placing the camera in the world and twisting it "up", so take its inverse
- Second part is the parents composed transformations (hierarchical modeling)
- Third part is the object's local transformation, in the order of:
  - $\mathbf{T}_{obj}$ is the translation
  - $\mathbf{R}_{obj}$ is the rotation
  - $\mathbf{S}_{obj}$ is the scaling
- $\mathbf{x}_{obj}$ is the object's local coordinates value

3D Drawing Concepts

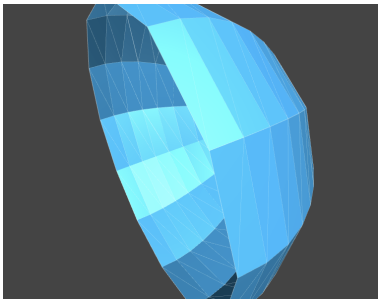
Geometry & Meshes

Meshes are collections on triangles that form (or approximate) 3D shapes. It is a combination of *geometry* (the shape) and *material* (color, lighting properties). We use triangles because they are efficient, flexible, and maps well to hardware.

Meshes have *vertices*, connected together forming *faces* (triangles) We use the *index-set representation* to represent faces. The order of the three vertices of a face matters because that determines the direction of the normal of the face.

Good meshes should have:

- Consistent handedness
- Avoid crackings
- Avoid T-junctions



A mesh has:

- Vertex properties, combines over the mesh using the *Barycentric coordinate interpolation*
  - E.g., interpolated color of vertices for any position on face: $P_{color} = \alpha A_{color} + \beta B_{color} + \gamma C_{color}$, where (α, β, γ) is the Barycentric coordinate of the point
  - Many other properties can be interpolated as well
- Face properties, constant over a face
  - E.g., a constant color of a face

To store the geometry topology, we could use different techniques:

- Polygon group
- Vertex sharing (index-set representation)
- Uniform patterns when appropriate:
  - Grids
  - Strips (edge connected triangles) / Fans (triangles sharing one same central vertex)

> The normal direction of a face determines which way is facing "outwards".
>
> - THREE uses *backface culling* by default - if the normal is facing away from the eye, you cannot see the face
> - Use `THREE.DoubleSide` to be able to see things facing backwards

Geometry can be built in a more general way by using *shape of revolution*: we define a function that describe a curve, then define shapes along the curve and interpolate to get a geometry. This gives us tubes, sweeps, lathe cones, ... Like the Chakin corner-cutting algorithm for 2D curves, we have schemes for dividing triangles to get smooth 3D geometries (Slides #26):

- Butterfly scheme: $v = (\frac{1}{2} - w)a + (\frac{1}{8} - 2w)b - (\frac{1}{16} - w)c + wd$
- Loop scheme (like B-spline in 3D)
- Catmull-Clark scheme of quads

Lighting & Material

Lighting implies the physical process of how lights take effect (though not necessarily need to simulate the real-world process to compute the effects). Another term, *shading*, is a more general term of computing the color of a pixel.

Light starts at a source, interacts with an object, and arrives at our eye.

- A simple local model of how photons go when hitting an object:
  - It could be absorbed, perfect mirror bounced, or affected by the micro-geometry
  - Depends on the *material* of object and the direction of the incoming light
  - Formally, we use *Bi-Directional Reflectance Distribution Functions* (BRDF): given an incoming direction and an out-going direction, what is the probability (amount of light) that happens
- Global lighting considers interactions among objects (the entire scene), e.g., occlusions, shadows, reflections, spill/bleed

Local lighting largely depends on the object's *material* properties. We consider the lighting model:

- *Emission*: things give off light
- *Ambient*: lights from everywhere
- *Specular*: direct reflection
- *Diffuse*: rough reflections

Lambertian materials scatter light into all directions:

- Doesn't matter where you look from
- Direction of light coming in does matter: the more "direct" the point is facing the light, the stronger diffusion
  - $r_{\text{diffuse}} = \mathbf{n} \cdot \mathbf{l}$, where $\mathbf{n}$ is the normal of object surface at the point and $\mathbf{l}$ is the light direction
  - $color = (\mathbf{n} \cdot \mathbf{l}) \cdot c_{light} \cdot c_d$, where c_{light} is the color/intensity of light and c_d is the color of the material

Shiny materials are specular:

- An ideal specular material is a perfect mirror
- In reality, gradually falls off as we get away from the optimal direction
  - $r_{specular} = (\mathbf{e} \cdot \mathbf{r})^p$, where $\mathbf{e}$ is the eye vector, $\mathbf{r}$ is the reflection vector, and p is the *shininess* factor
  - $r_{specular} = (\mathbf{n} \cdot \mathbf{h})^p$, where $\mathbf{n}$ is the normal of object surface at the point and $\mathbf{h}$ is the half-way vector between eye vector and the incoming light vector
  - $color = (\mathbf{n} \cdot \mathbf{h})^p \cdot c_{light} \cdot c_d$

The **Phong material lighting model** in one formula looks like:

$$color = c_e + c_a l_a + \sum_{l \in lights} ((\mathbf{n} \cdot \mathbf{l}) \cdot c_{light} \cdot c_d + (\mathbf{n} \cdot \mathbf{h})^p \cdot c_{light} \cdot c_d)$$

- *Metalness*: more specular or not
- *Roughness*: amount of specularity

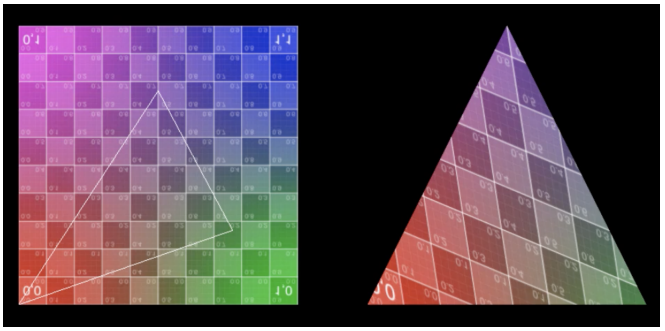
Different strategies for computing the color for an object in scene:

- Flat shading: once per triangle
- Gouraud shading: once per vertex and interpolate colors
- Interpolate normals: once per pixel (used today)

Texturing

Textures (纹理) help us have more information in a triangle (other than just three points + a face). The basic texture mapping takes the coordinate of a point on the triangle, uses an image (color map) to lookup the coordinate to color, and use that color as the color for that point.

We typically use **UV coordinates** on the triangle: specify a pair (u, v) value for each vertex, then the UV value of a point is the Barycentric interpolation of vertices UVs.



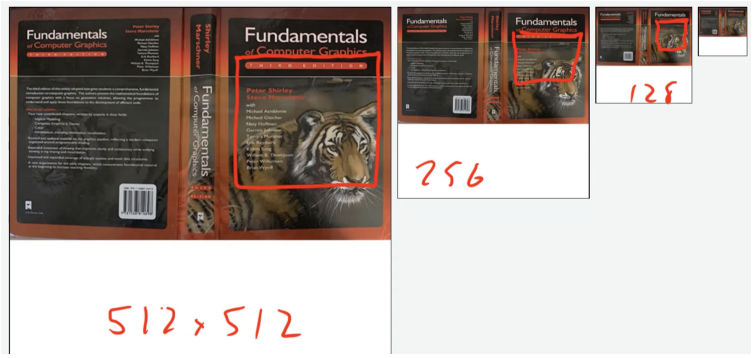
- Left is the UV plane, both going from 0 to 1
- Mesh with multiple triangles (next to each other) can map nicely onto the UV plane texture image
- What if a point maps to UV outside of $[0, 1]$?
  - *Clamp* wrapping: just chop off and map to min/max value
  - *Repeat* wrapping: round over from other side
  - *Mirror repeat* wrapping: mirror the texture map

> Loading and storing textures is expensive, so we want to use fewer texture maps and map as many triangles as we can to one texture map.

The final color of a point is the composition of {texture, material, lighting}.

Since the UV map is continuous but pixels are discrete, it can be very tricky when we try to map a continuous thing to discrete canvas image (*magnification* & *minification*). Instead of thinking of pixels just as squares, better solutions are:

- On magnification:
  - Nearest neighbor: think of colors living on grid points. each pixel takes the nearest neighbor color
  - *Bi-Linear* interpolation: combine the 4 neighboring colors
  - (Using shader anti-aliasing, e.g., `smoothstep` with `fwidth`)
- On minification:
  - *Filtering*: averaging the colors a pixel covers
  - *Summed Area Table*
  - *Mip-Maps*: approximate filter as a square - precompute multiple sizes of squares - pick the two correct sized squares and interpolate at runtime
    - Considering anisotropy (各向异性), if the triangle is mapped to a really stretched region on UV map, using a square approximation might be way too big; In this case, we may want to use areas that are more appropriate; THREE supports anisotropic filtering by breaking into smaller squares - better look, but more lookups
    - *Tri-linear* interpolation: use images of smaller and smaller size, lookup in-between two images (one just little bit too big and one just little bit too small), each using bi-linear, and combine (in total 8 lookups)



Occlusion & Visibility

Nearer objects cover farther objects from the camera's perspective. To figure out what color should eventually appear on each pixel on screen:

- *Painter's algorithm*:

```
sort triangles in camera Z order
for triangle from farthest to nearest:
    draw all pixels of the triangle
```

  - Inefficient - need to sort
  - Inefficient - no matter what, draws all the pixels in all the triangles (figuring out the coloring, lighting, shading...); but for many pixels, only the nearest triangle covering that pixel will eventually win
- *Z-buffer algorithm*:

```
for pixel (x,y) in screen:
    for triangle in any order within [Z_near, Z_far]:
        pz = ZB(x,y)
        if Z_triangle < pz:     # draw only if nearer than nearest seen
            draw the pixel
            ZB(x,y) = Z_triangle
```

  - *Z-fighting* - if two triangles have very close camera Z value or have ties, due to numerical errors they may fight on the test; In this case, the ordering of triangles matters
  - Semi-transparent objects - ordering also matters; need transparent objects last
  - *Aliasing* decision - if two triangles both cover only part of a single pixel, need a way to decide

Web APIs Potpourri

Javascript

See Javascript document here: [the MDB JavaScript doc](#).

One thing that is most useful for making little animations in the web browser is to use the `requestAnimationFrame` feature:

```
function updateFrame(timestamp) {
    ...
    window.requestAnimationFrame(updateFrame);
}
window.requestAnimationFrame(updateFrame);
```

Canvas

HTML Canvas Doc: [the MDN Canvas API doc](#).

Things to note about the HTML Canvas API:

- Coordinates:
  - By default: `x` & `y` goes out from the top-left corner, relative only to the canvas element itself
- Doing `.fill()` vs. `.stroke()`

- Context `.save()` & `restore()` behave as a stack
- Path sides when filling a non-simple path:
  - *Even/Odd*: pick any point, go any direction, and count the number of crossings of paths; even → outside, odd → inside
  - *Non-zero winding*: count the "loops" around a point, +1 for clockwise, -1 for counter-clockwise; non-zero → inside, zero → outside

SVG

SVG Doc: the MDB SVG API doc.

See a good set of SVG tutorials here: CS559 SP2021 SVG Tutorials.

THREE.js

THREE.js is a *scene-graph* API just like SVG, but is built for rendering 3D graphics and can manage the graphics hardware from browser through backends like WebGL.

THREE Doc: the THREE.js official doc.

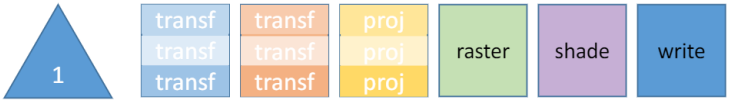
Things to note about the THREE.js API:

- THREE renderer creates a 2D canvas element for us where the picture goes into, and we typically append that element into the HTML DOM
- Coordinates:
  - `x` goes right
  - `y` goes up
  - `z` follows the *right-handed rule*, so goes out of the screen
- Animations with THREE:
  - Transformations (matrices), material properties, and lighting properties are easy to change
  - Changing mesh geometries, material types, and lighting types are hard - they require sending large data to the hardware / recompiling the shader
- State of vs. transformation on an object:
  - `cube.position.x = 5` just sets the position forcefully
  - `cube.position.x += 5` just changes the position relative to its only position in the world coordinate system, but is not affected by its local context
  - `cube.translateX(5)` applies a transformation to the object, affected by its local context (where is its local x-coordinate pointing to)
  - In summary:
    1. State: `matrix`, `position`, `scale`, `quaternion`, `rotation`
    2. Transformation: `applyMatrix4`, `translate`, `applyQuaternion`, `rotate`
    3. Special (setting an absolute orientation): `lookAt`, `setFrom`, `vUp`
- Each object can have children; Hierarchical modeling should better use *groups*

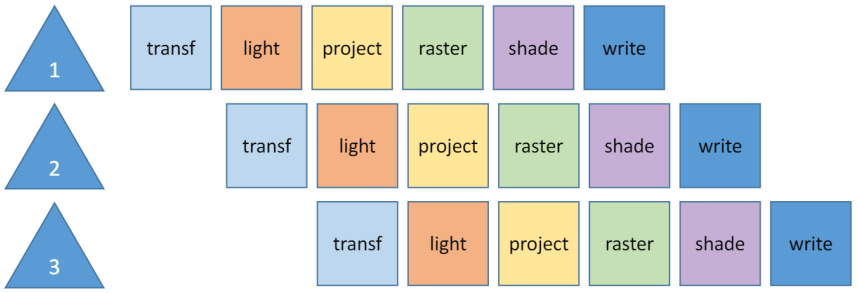
Graphics Hardware & Shaders

Graphics chips (GPUs) exploit the following two features intensively to draw triangles fast:

- Massive parallelism: triangles/vertices/pixels are mostly independent with each other, so in each processing stage, parallel the processing of independent entities



- *Pipelining* parallelism: well-designed and programmable (flexible) pipeline

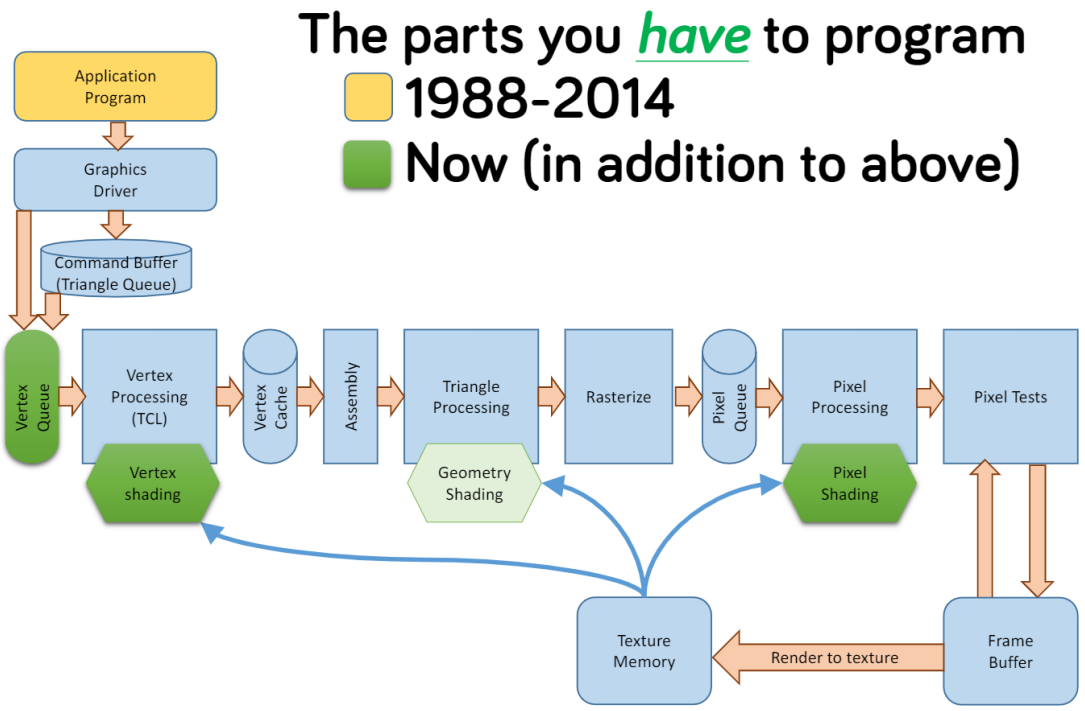


Recall the entire **rendering** process for 3D drawing:

1. *Modeling* - put triangles in 3D world
2. *Viewing* - calculate position relative to the camera eye
3. *Clipping* - decide if triangle is not too far/near so it should appear on screen
4. *Projection* - transform to the 2D screen space
5. *Visibility* - decide how triangles occlude one another
6. *Rasterization* - convert triangles to pixels
7. *Shading* - color each pixel according to the triangle vertices' *texturing* and (local/global) *lighting* properties

Graphics Pipeline

Just like CPUs, GPUs do pipelining intensively, but in a more programmable and flexible way, controllable through small programmable parts.



The two must-have programmable parts are:

- *Vertex shader* (per object info + vertex with info → vertex with more info & *screen-space position*)
- *Fragment (i.e., pixel) shader* (per object info + fragment with interpolated info → fragment with more info & *color*)

Shader Programs

Shaders are written in a special category of languages called *shading languages*. Examples include:

- Pixar Reyes, RenderMan
- GLSL for OpenGL
- GLSL-ES for WebGL: Reference card

They include the actual shader program in the shading language and a host API, say from Javascript, to send the program to the hardware. The shader program must appear as a pair of (vertex shader, fragment shader).

```
// Vertex shader example:

uniform mat4 projectionMatrix; // constant object info, sent by THREE
uniform mat4 modelViewMatrix;  // ..
attribute vec3 position;       // per vertex attribute, sent by THREE

varying vec4 gl_Position;      // output attribute

void main() {
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
}
```

```
// Fragment shader example:

uniform vec3 ourColor;         // customized uniform, sent by us through JS

varying vec4 gl_Position;      // per fragment property, interpolation of the vertex shader's output

void main() {
    gl_FragColor = vec4( ourColor, 1);
}
```

In THREE, shaders are defined as part of a material. We can apply textures now in two ways:

- Image based texture, like we have covered before
- **Procedural texture**: make the (fragment) shader program compute the texture

Please find techniques, tricks, and suggestions on writing shader programs in workbook 11.

Random Noise

In CG, we typically do not want true randomness but just *pseudo-randomness*, so that we can reproduce the same image every time. Methods for making noise include:

- Simple ones: sin noise (`fract(sin(u_sampled)*1000000.0)` with anti-aliasing)
- Combine multiple frequencies
- *Perlin noise*
- More complicated ones

Advanced Texture Mapping

Shaders enable us to apply several advanced texture mapping techniques:

- **Displacement map**: change the geometry of the object according to the map, typically the height along normal
  - Can only happen in the vertex shader
- **Normal map**: use RGB color as a 3-tuple to change the normal direction and magnitude on individual locations on the mesh. This can give us fake shapes
  - E.g., interpolating the normal so that an imperfect cylinder (made up of 12 sides) looks like a circular cylinder under lighting
  - Can happen in vertex shader, but typically done in fragment shader
- **Bump map**: a simplified version of normal map, fakes the effect of each point on the surface moving towards/away along the normal direction to make it look bumpy
  - Can happen in vertex shader, but typically done in fragment shader
  - Does not look good if we are close to the object and looking from the edge

We often use a big "box" with texture inside to surround the viewpoint to make the far-away background. Since they are so far away, distance (position of the observer) does not matter too much, but *orientation* (normal vector) does matter. This box is called a **skybox**:

- Cube skybox (*Cubemaps*)
- Sphere skybox (*Equiectangular maps*)
- Cylinder skybox (*Panoramic maps*)

The skybox is often loaded as a *static* **environment map** of an object to help compute *reflections*.

To get reflections about other objects, or if the mirror object is moving around, it requires *dynamic* environment maps. We could do that by *multi-pass rendering*:

1. Draw once to get information about all the objects, use a cube camera to take a picture about other objects
2. Draw the second time to actually make the reflective object

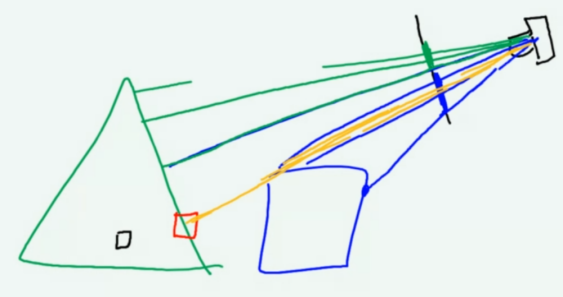
Shadows & Real-Time Rendering

Multi-pass rendering is an example of *real-time rendering*:

1. In earlier phases, draw the objects and compute some dynamic texture maps (e.g., dynamic environment maps, or shadow maps described below)
2. In the final phase, color the objects using the advanced dynamic maps computed

To get advanced, non-local lighting effects like *shadows*, we add yet some other layers of texture mappings:

- Pre-computed ("baked") lighting: **ambient occlusion map**
- Use the first pass of rendering to take a picture of which points can a light cast on, store this as the **shadow map** per light source; In the second pass, for every point on the object, look up if this point appears in the shadow map or not, if yes $\rightarrow$ not shadowed, if no $\rightarrow$ shadowed

  The value stored in shadow maps are typically Z values: if the point's Z value $>$ the Z value in shadow map, it means there is something blocking this lighting source.

> Dynamic shadows maps in 3D scenarios is often referred to as the *ray-tracing* technology.

In THREE, to enable shadow maps:

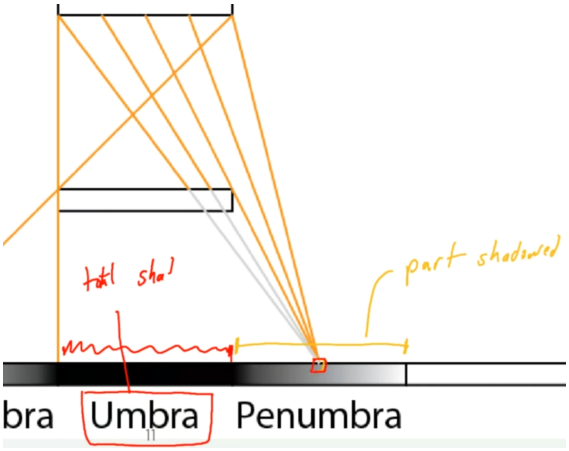
- Setup lights to cast shadows $\rightarrow$ they will make shadow maps
- Setup objects to cast shadows $\rightarrow$ they will be rendered in all passes
- Setup objects to receive shadows $\rightarrow$ they will access all the shadow maps in the final pass
- Setup render to do shadows $\rightarrow$ there will be multiple passes

Advanced Rendering

Some advanced hacks & optimizations used in real-time rendering:

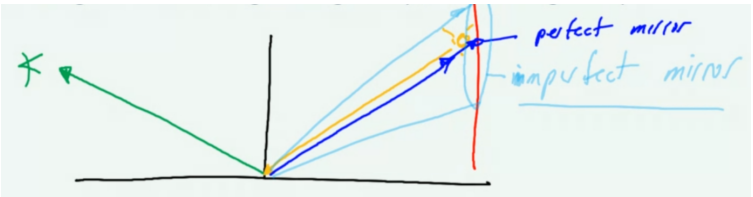
- *Hard* shadows vs. *Soft* shadows from *area light sources*:

  Hard to simulate with shadows maps, but a workaround might be to simulate the area light source as a collection of point light sources - this is very inefficient

- Filtering a range of the environment map in reflection:

  As "shininess" goes down, size of filter area goes up



- **Deferred shading/rendering** to avoid making the fragment shader stage of the pipeline too heavy and to avoid shading unseen points:
  1. Don't shade fragments when we draw them, just use Z-buffer for locations
  2. Store information needed for shading/coloring in a *geometry buffer* (G-buffer)
  3. In yet another stage, compute the colors; If multiple points from multiple objects appear in the same pixel on screen, only the nearest one will be done shading computation (unless there is transparency)
  4. If lights do not cover the entire screen, or if we only have limited memory, we can break the screen into smaller tiles and do the shading computation tile-by-tile. This is called *tile-based deferred rendering* (TBDR)
- Animation by **deformation** (transformations on a base mesh): when changing the geometry, we send transformations to the GPU instead of the whole new geometry, so we reduce the data size we need to transfer from main memory to GPU memory
  - Shape interpolation (*Morphing*): $p = w_1 p_1 + w_2 p_2 + w_3 p_3$, send all meshes to hardware, each frame only change the weights
  - Can be non-linear: twist, bend, lattice (grid), free-form (FFD), cages (build shapes inside cages using Harmonic coordinates, then control the cages), skeletons, ...
    - *Cages* & *Skeletons* of rigid pieces are common in interactive graphics and games
    - **Skinning** is the process of covering the gaps at junctions to make it look more natural; *Linear-blend (smooth) skinning* means assigning a vertex at junction to be α position of bone1 + $(1 - \alpha)$ position of bone2

Rasterization

Rasterization (栅格化, mapping triangles to pixels) is actually a hard problem. The basic problem is deciding whether a pixel needs to be colored for a triangle and what color it should be. Some of the topics have been covered in this section.

> "A pixel is not a little square. It is a sample point." -- Prof. Michael Gleicher

Methods:

- Cover sample point? -- handles edges consistently, but triangle could get lost between pixels
- *Scanline algorithm*:

```
for every row of pixels:
    find left and right boundary of triangle coverage
    fill those pixels in between
```

- In today's graphics hardware: using Barycentric coordinate because this is massively parallel

```
for each pixel:
    compute barycentric coordinate relative to triangle
    decide if in or not (having negative coord or not)
```

- Drawing lines with the *Brezenham midpoint algorithm*: pick closest pixel for each column

Rasterization also affects our general **anti-aliasing** (抗锯齿) effect. We prefer blurry things over sharp / jaggie edges, so we do not want each pixel to be exactly the same color of the triangle, but instead some interpolation of closest triangles.

We do *super-resolution*: breaking each pixel into more smaller sample points (4x, 8x, 16x, ...), do sampling on each one using the above methods, then average over them to produce the eventual color for that pixel.