Computer Graphics

What this talk is **not** about:

How to add graphics to your project
How to use Photoshop
How to make a GUI in Java
… or any other 'how to do X'-type of topic

What this talk **is** about:

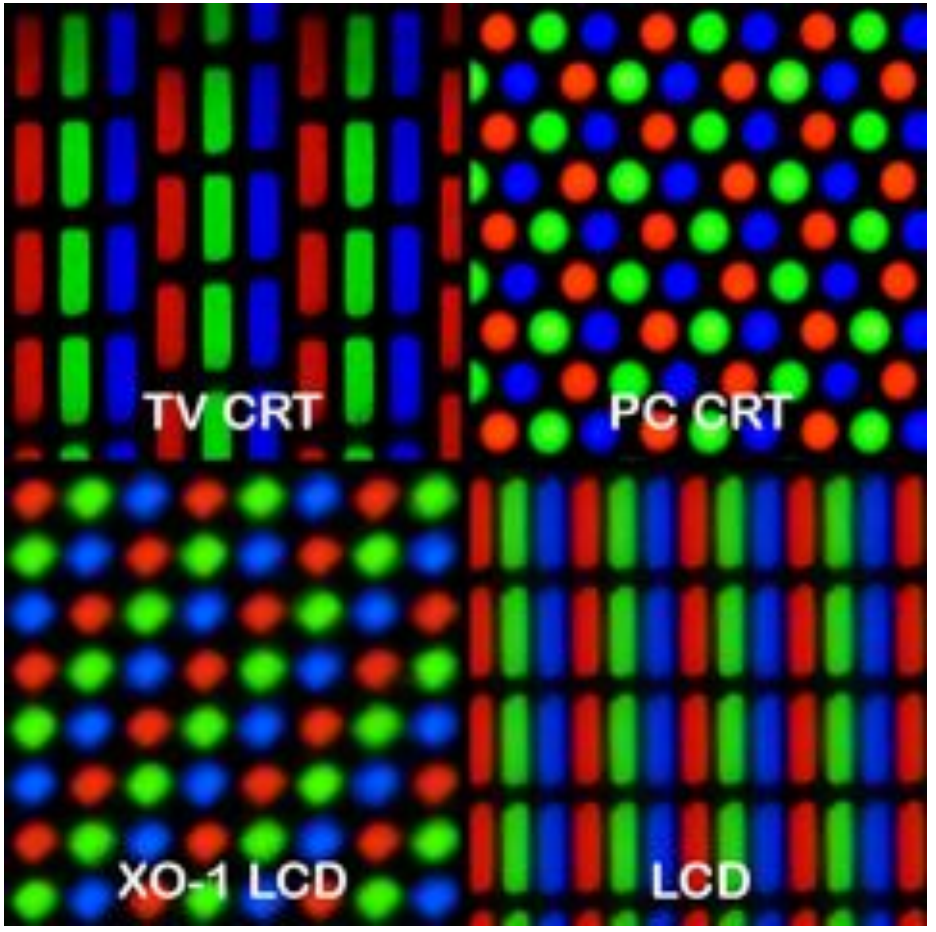
Computer graphics is a pretty big field that has been with us for a long time. It involves all sorts of awesome stuff!

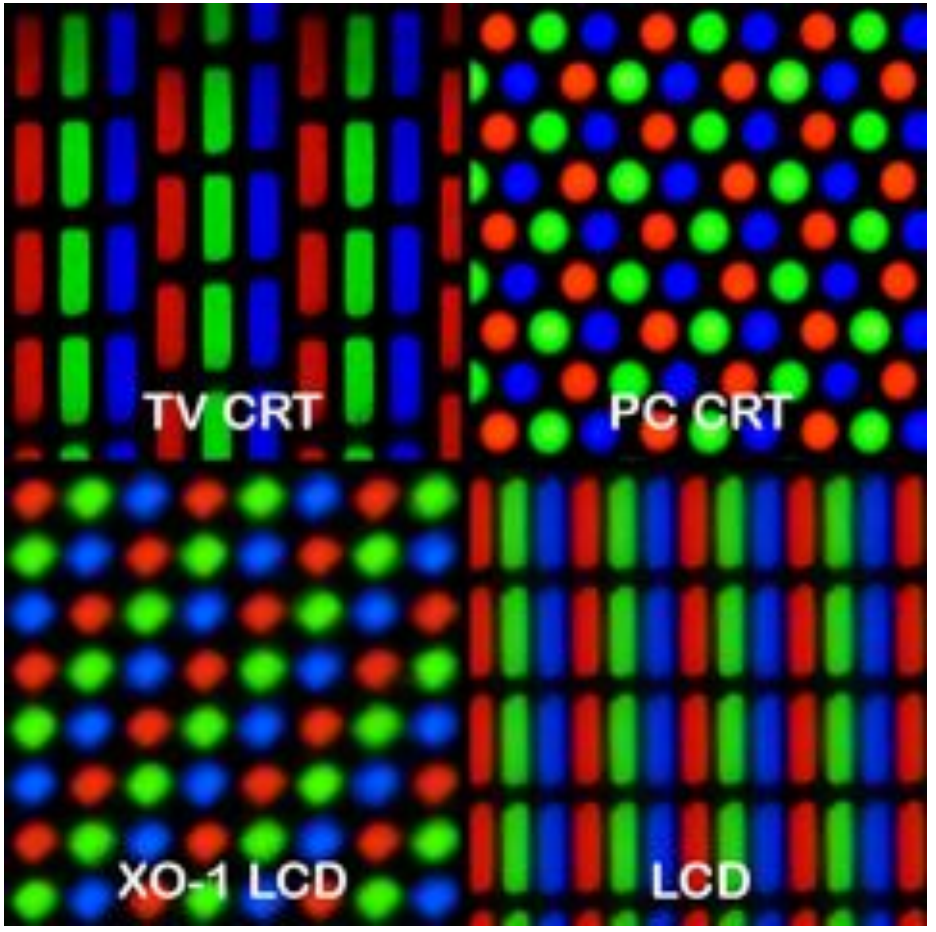
Mathematical Modeling
Data Structures
Algorithms
Custom-built hardware

Human-Computer Interaction
Design
Simulation & Modeling
Movies & Games

Concept #1:

Pixels are the (visual) bridge between computers and people.





Pixels!

These are the boundary between the computer's electronic world and our world of photons.

Each pixel is a trio of *red*, *blue*, and *green* elements that form a single *picture element*, or *pixel* for short.

Here's a single pixel.

A computer's graphics system is dedicated to 'drawing' these things. From software's perspective, they have to be drawn about one time every 16ms—that is 60 times per second.

On a Macbook Pro with a Retina Display, there are over 4 million pixels.

So: 16 milliseconds to draw 4,096,000 pixels. Note that.

Each pixel's red, green, and blue components can have a variable intensity. They can be entirely off (that's black), or all entirely on (that's white). Say the red and blue are at 50% intensity and the green component is off—that's dark purple.

Since computers are digital, we need to use numbers to encode how intense each component will be. We can use as many bits as we want, but on modern hardware we typically use either 8 bits per component (==24 bits per pixel).

Pushing Pixels

OK, so we have pixels, and they'll be drawn pretty frequently, and there are a lot of them, and… they take 3 bytes each. Sounds good. So what should we have those pixels actually *do*?

I don't know. What do you want them to do?

Watch a movie?
Show your term paper?
Render an architectural model?

Would you like to play a game?

Concept # 2:
We model things to draw using data structures.

By modern standards, the NES *Super Mario Bros.* seems pretty antique. But this is a good place to start understanding how modern graphics work, because the fundamentals are basically the same.

Games have all the same problems that more boring software does: it has data structures, algorithms, procedures, memory, input and output, and so on.

We'll concern ourselves with the parts of that dealing with making pretty pixels on the screen.

Each pixel has:

- a location
- a color

This Mario sprite is 256x256 pixels. There are four colors in this one: red, brown, beige, and *clear*. OK, so in addition to red, green, and blue, we have *alpha* which represents how opaque or transparent the pixel is.

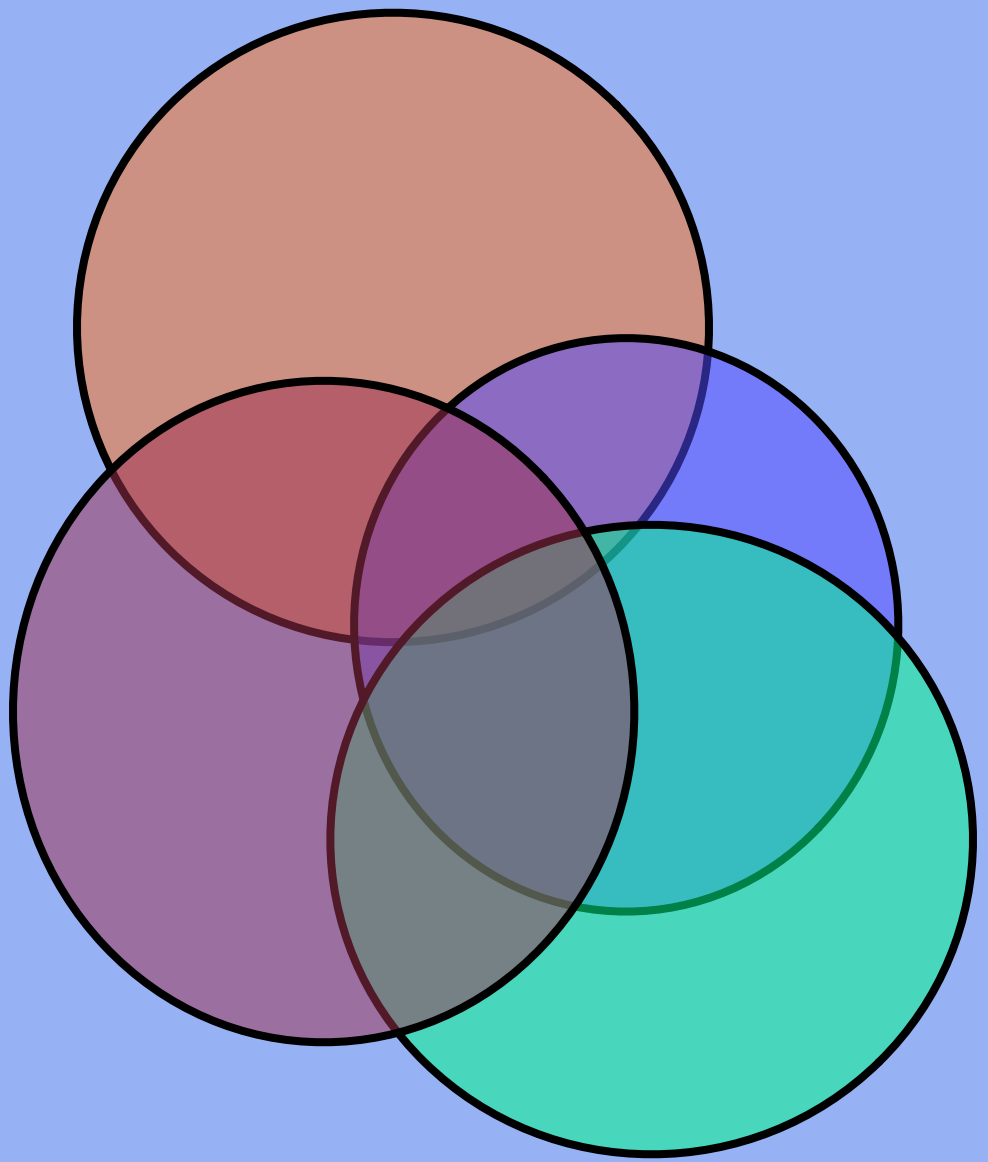
We draw things differently depending on what is going on. We might have a bunch of different versions of Mario that we need to draw depending on if he's jumping, crouching, climbing, or eating magic mushrooms.

So we need:

(1) Data structures to record the game state (where is Mario, what's he doing)
(2) Data structures to render game elements depending on #1.

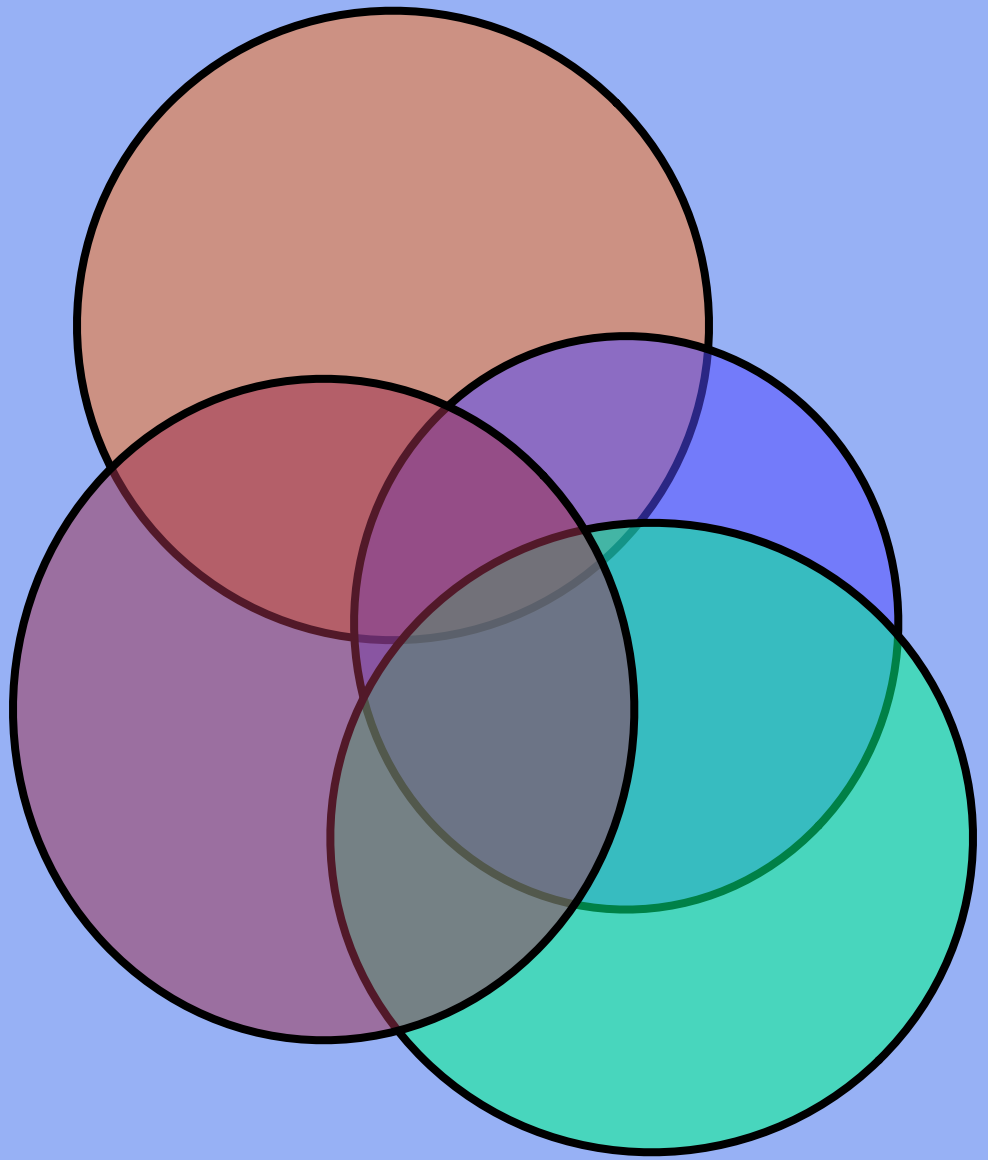
Concept # 3:

Colors are numbers, and we can do all sorts of groovy math with numbers.

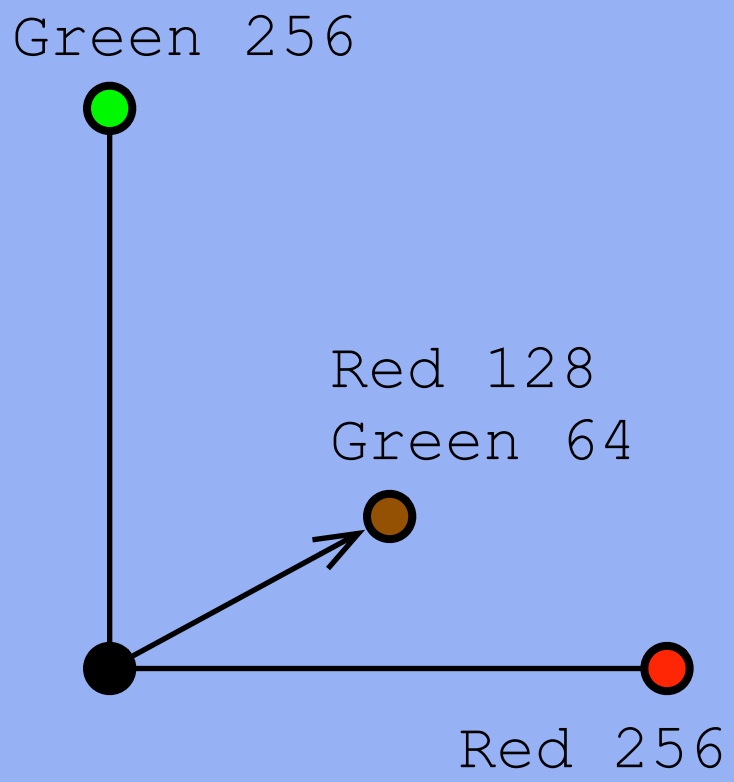


The circles over yonder are all filled with colors. Since those colors are translucent (their alpha value is somewhere between off and on), that means we should be able to peer through them to see what is below.

But what color should we see? And how do we calculate it?



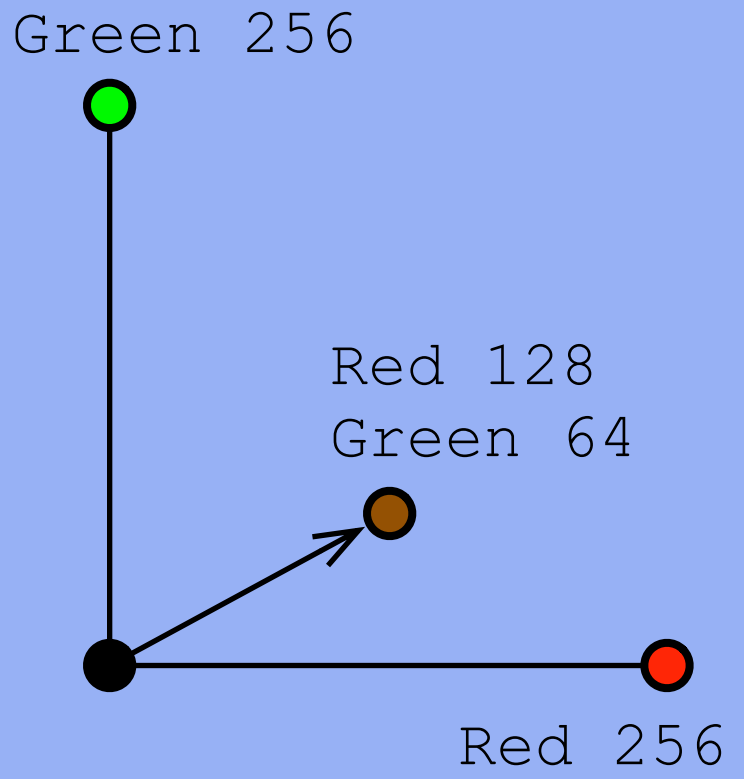
Since each pixel is defined by several components (R,G,B, and Alpha), we can use a mathematical *vector* to represent it, and do math.



A vector is just a bunch of values that are stuck together that tell a bigger story. This is an example of a 2-dimensional vector with red in one dimension, and green in the other. If a color has neither of these, it is black. If a color is halfway along the red dimension and a quarter way along green, the result is brown.

This is just a 2D vector, and real colors work with 3D. But to keep things simple, you might write this color **C** like this:

C = <128, 64>

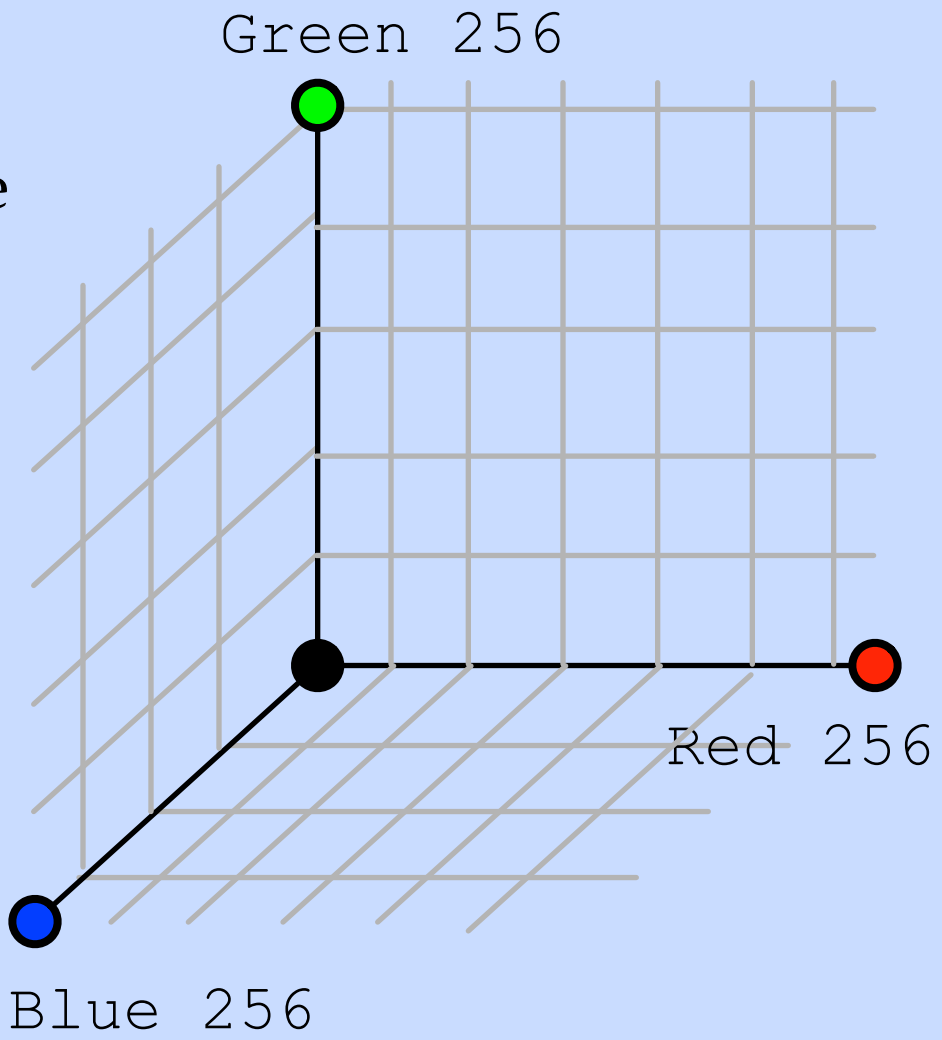


There's lots of other colors in this weird Red/Green space, and each component is independent of one another.

(In fact there are 256^2 possible colors in this color space.)

A more realistic color space includes the Blue component. We can use this to specify the intensity of real pixels.

C1 = <40, 0, 240>
C2 = <0, 190, 177>
C3 = <192, 192, 192>
… and so on

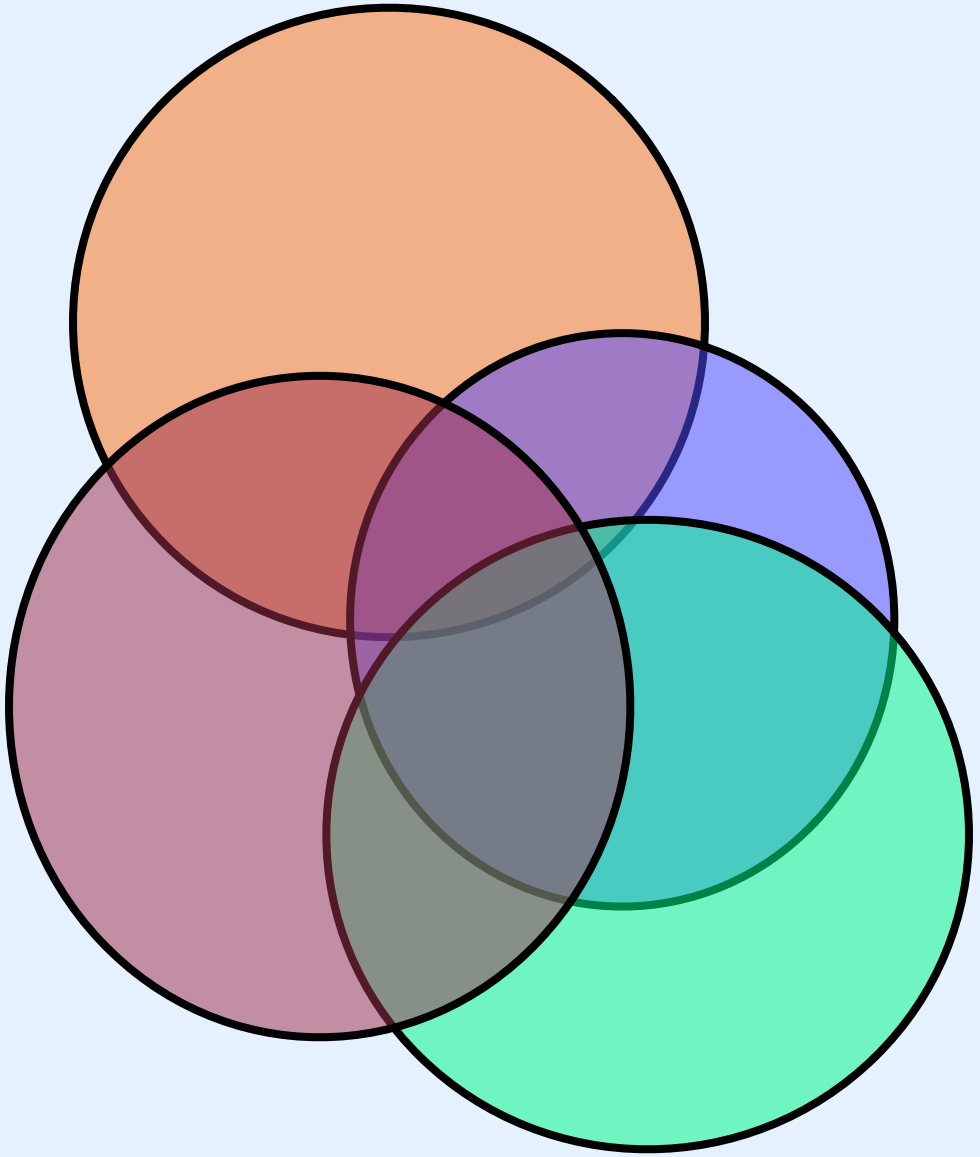


This (terrible) diagram represents a 3D color space. But we might also have *alpha*, which would be represented with a 4D space. I don't know how to draw that.

Back to this problem. If we have one semi-transparent color sitting on top of a couple other semi-transparent colors, what color should the pixel be?

This problem is called *compositing*, and there's many ways of doing it.

Usually for this kind of math, the components of the color vectors are converted to floating point numbers in the range of 0 to 1.



Concept #4:

Directions and points can both be represented with vectors. (And of course we can do groovy math with vectors.)

```
velocity = <0.73, -1.33, 3.84>
up = <0.0, 1.0, 0.0>
bad_guy = <-47.0, 50.4, -19.5>
```

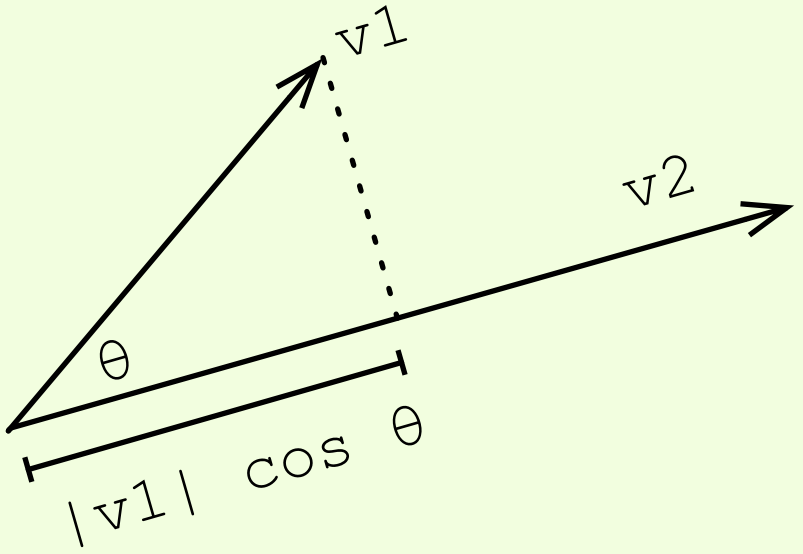
A Direction might be represented with unconstrained numbers (like *velocity*), or it might be constrained to be a *unit vector* (like *up*). Unit vectors are always 1 long.

A point looks exactly the same. The only difference is how we interpret it. Is the vector a direction? Or a point? (Or a color?—remember we can do that, too!)

The groovy math: You can get away with a LOT with simple addition and subtraction of vectors. Other operations include the *dot product* and *cross product*. These simple operations involve basic arithmetic—nothing more.

Dot products let you measure the angle between vectors.

Cross products let you find a third vector that is a right angle to the first two.



Find out more at a calculus class near you!

Concept #5:

Matrices can be used to store *transformations* in 3D space. They can also store the result of an arbitrarily long sequence of such transformations.

Make an *identity* matrix M that at the origin.
Want to rotate 90° about the x axis? *Multiply M with a matrix.*
Want to translate 80 pixels along z? *Multiply M with a matrix.*
Want to scale by a factor of 19? *Multiply M with a matrix.*

At every step, we multiply our transformation vector by some matrix that defines one of the main operations: rotation (turning), translation (moving) and scaling (zooming). We always store the result back in M, our transformation. *Easy peasy.*

Matrix transformations are (almost entirely) done with simple arithmetic: addition and multiplication, for the most part. There's nothing scary hard at all. It is just incredibly *tedious* to do by hand.

Fortunately: **computers!**

This is where hardware can step in to make life awesome.


```
It makes life awesome.
```

Concept # 6:

Many graphics operations can be done (1) in parallel with each other, and (2) using matrix and vector operations.

Parallel work:

Say I ask the class to compute this:

(5+9) * (14 / 3) + (19 - 12 + 4 * 3) * (3 + 2) + (9 * 13 * 2)

We can give each term (thing in parens) to a different person and have them solve their part independently, and *in parallel*. We only need to get their answers and combine them. We might be able to parallelize *that*, too.

That's parallel computing, and CG depends on it.

Lets say we are a graphics card and we're asked to take the mathematical model for this image above and calculate pixel values. All 4 million of them.

With two cores, we can divide this work in half. 2 mil each. With four cores, quarter the effort. 1 mil each.

With 1024 cores, *each has to do a mere 4000 pixels*. Modern graphics cards are beginning to have this many cores.

Matrix math:

Now that we have all these cores running in parallel with each other, what do they actually do?

GPUs are designed to do matrix math as fast as possible. They are 'domain specific hardware' in the sense that they are insanely good at one particular thing: transforming matrices of floating point numbers.

Next time you see a CG movie, thank a mathematician or electrical engineer.

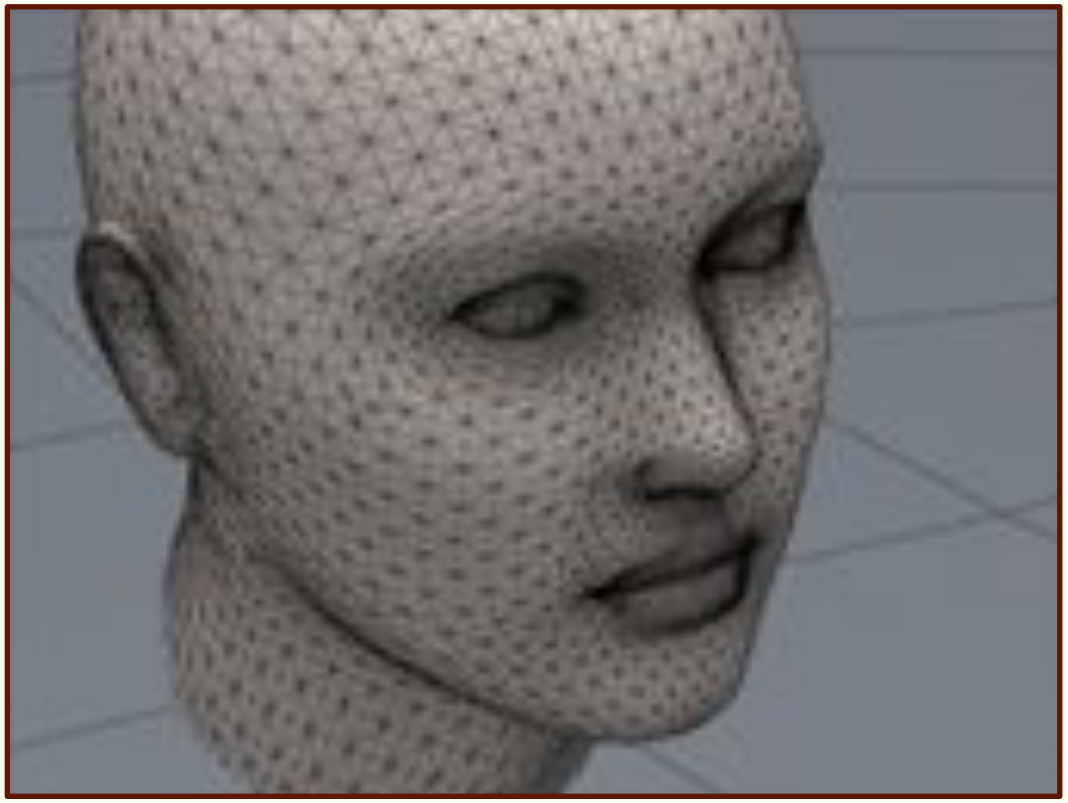
Concept #7:

CG models involve *vectors*, *meshes*, and *textures*.

Now that we have the math and hardware to do the basic CG operations, how do we make a game or movie? We need a couple more things.

A *mesh* is a common approach to store a bunch of points (which are *vectors*) and how they relate to one another. This gives us polygons.

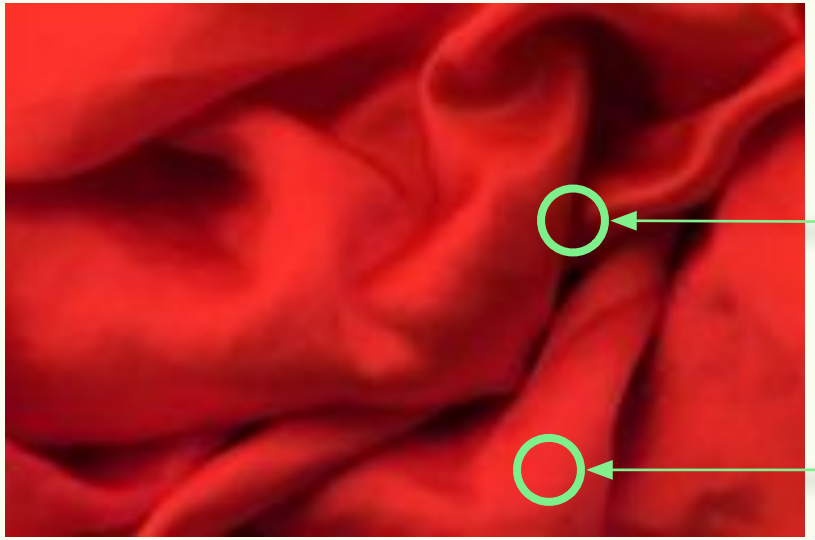
That's a mesh. →



Textures are just pixel data that we overlay on top of the mesh polygons. They usually represent the 'skin' for the model, like this:

We can use additional textures to augment the model with *bump maps* or *normal maps*. They represent how light should bounce off the model at each location. That's how you get neat reflection properties like below.

For example, say we are making a CG game and would like to model cloth. It would be easy *and totally boring* to render each part of the cloth with the same color. The cloth has folds and bends. Some parts are facing you, others face away.



These two pixels in here are the same cloth but the top is bent away, while the lower one is facing you. Notice the difference in their apparent color.

We could use each polygon's *surface normal* to compute the apparent color by comparing it with our viewing location. This is 1970s-style shading, and it works pretty well, but these days you only do that when you have old hardware.

Another way of rendering the pixel values could be to use additional textures like bump maps. To compute an apparent pixel value, compute the angle that light bounces off the cloth given each polygon's surface normal and the bump map texel.

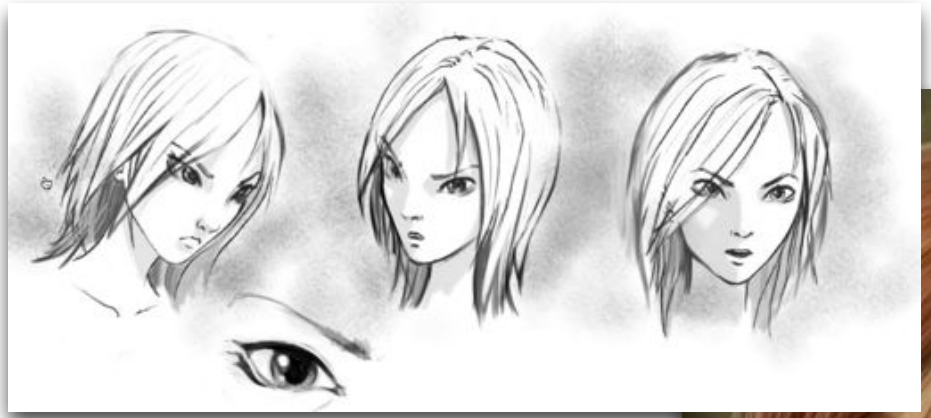
A *texel* is like a pixel, but is encoded in a texture. It can be treated like visual data (colors) or other data (directions, displacement, etc.)

Using textures to determine how light bounces off things is pretty powerful, and it can be done using the relatively simple math that modern GPUs do so expertly, in parallel.
Now that 16ms redraw rate doesn't even seem challenging!

Obviously, CG isn't just
about the technical stuff.

This brings us to…

Concept #8: designing *with* and *for* computer graphics



Sintel (2010)

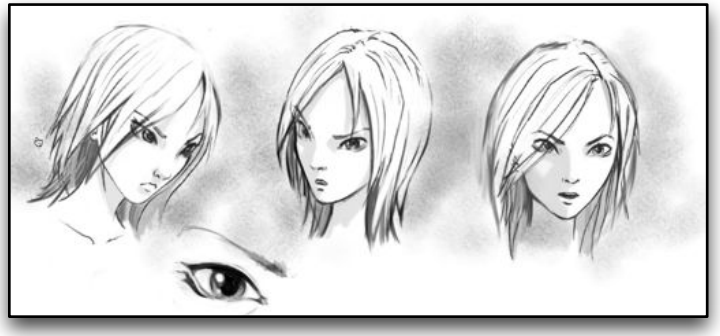
Blender Foundation

(it is really sad)

Artists tend to start out by making freehand sketches on paper, all old-school style. Don't underestimate the necessity and utility of this step. Designers know what I'm talking about. But for all you engineers out there: *pay attention to this because it applies to you too*.

You've doubtlessly used something like Photoshop or Illustrator. These are pixel- and vector-art programs and are really common in the design process.

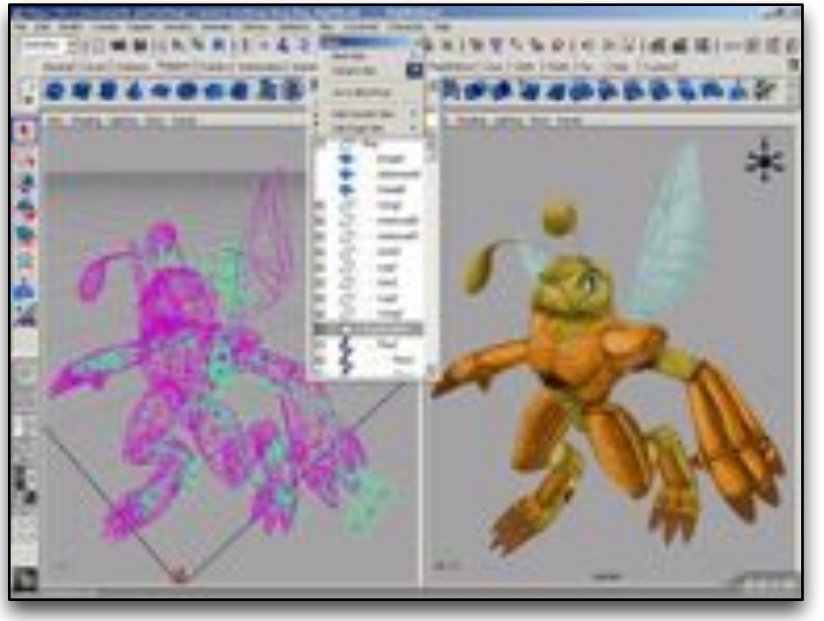
paper --> pixels



-->

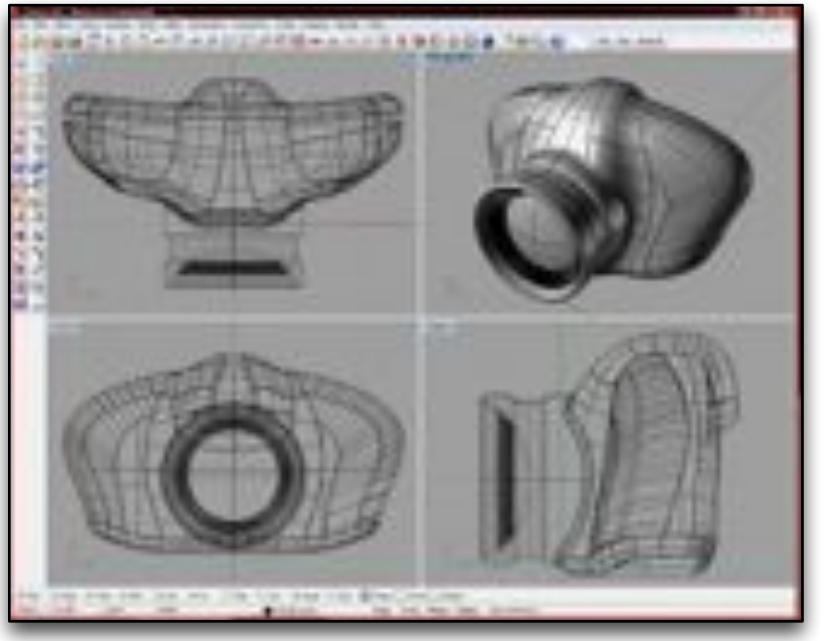
There are loads of other modeling programs for doing 3D computer art:

— Mudbox
— Cinema 4D
— 3D Studio Max
— Maya



… and then there's even more modeling programs for gameplay or architecture/engineering models:

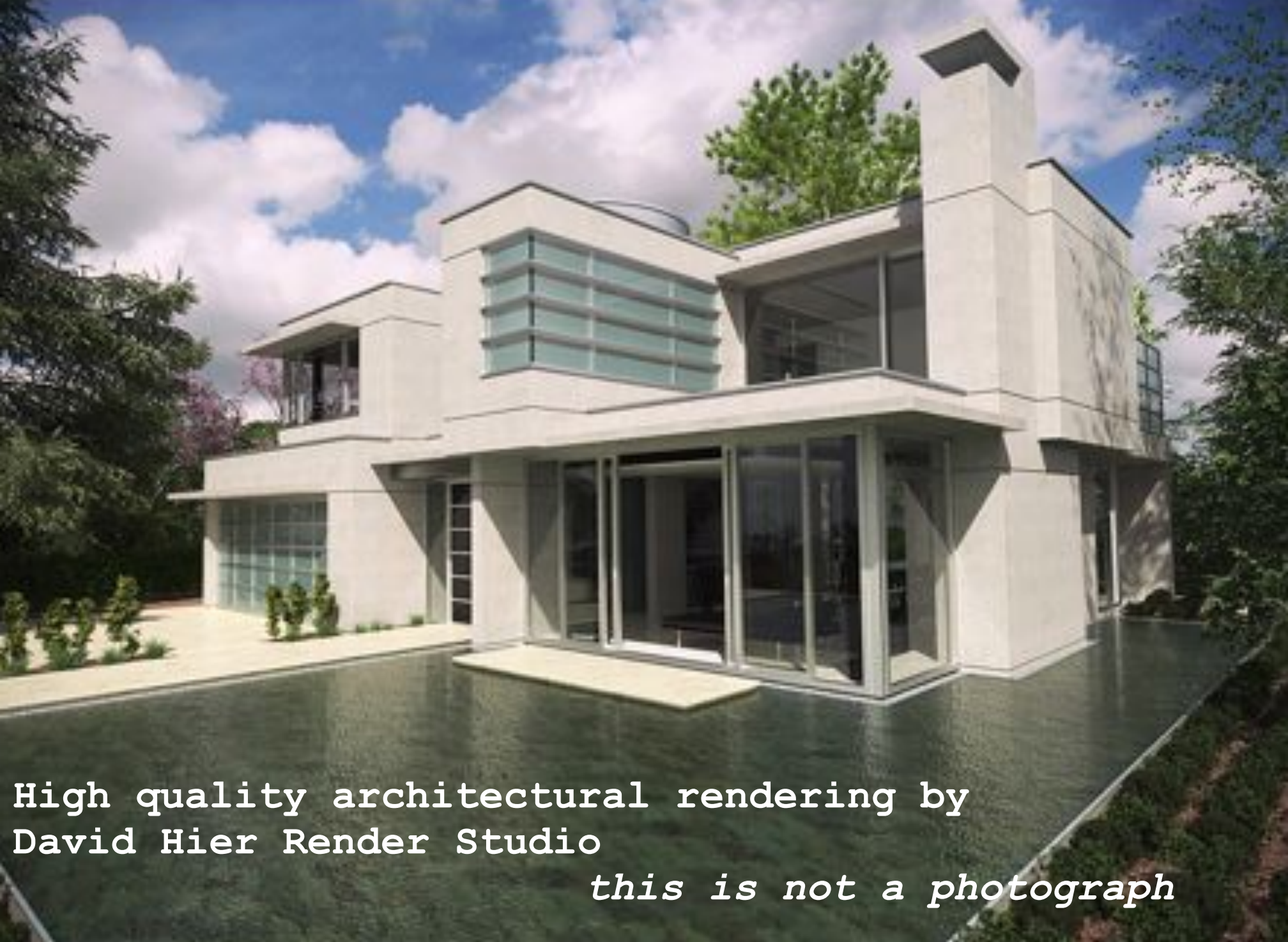
— Unity 3D
— Revit
— SketchUp
— Rhino



There's lots and lots of things to consider when designing anything, including CG. Here's some CG-specific constraints:

How fast does it need to render? *Monsters University* by Pixar took 29 hours CPU hours to render each frame in the final cut. By contrast, a typical 3D video game should render each frame in less than 16ms in order to appear smooth.

How good does it need to look? Movies, games, and engineering renderings all have different requirements for how good they look. Rendering quality depends on aspects like polygon count, hardware (number of cores, amount of video memory), and how much special processing must take place.


High quality architectural rendering by
David Hier Render Studio
this is not a photograph

The rendering in the previous slide probably took hours to compute (I'm not exactly sure how long). But if you're *designing* in a modeling tool you'll go absolutely insane if the realtime rendering takes that long. So: your tool will have a setting to render in *interactive speed* as well. Won't look as good, but you won't go crazy while you wait for pixels to show up.

Concept #9: Being clever about how to model things.

For example: Hair!

For a fast render (interactive speed) you'll probably just use a texture.

For a slow but high quality render, you might model individual photons bouncing around individual hairs.

There are people who specialize in rendering these things: hair, water, fire, vapor, eyes, lips, *you name it*. How'd you like to be the world expert on rendering *nostrils* and get paid goo-gads of cash?

How would you model rendering hair?

You could do individual hairs. Or clumps of them.

How does hair absorb/reflect light?

Does it matter what angle the light is coming from?

What if there are multiple lights?

What about diffuse light?

Can we change hair color easily?

Can hair move around in the wind? Or as the character turns her head?

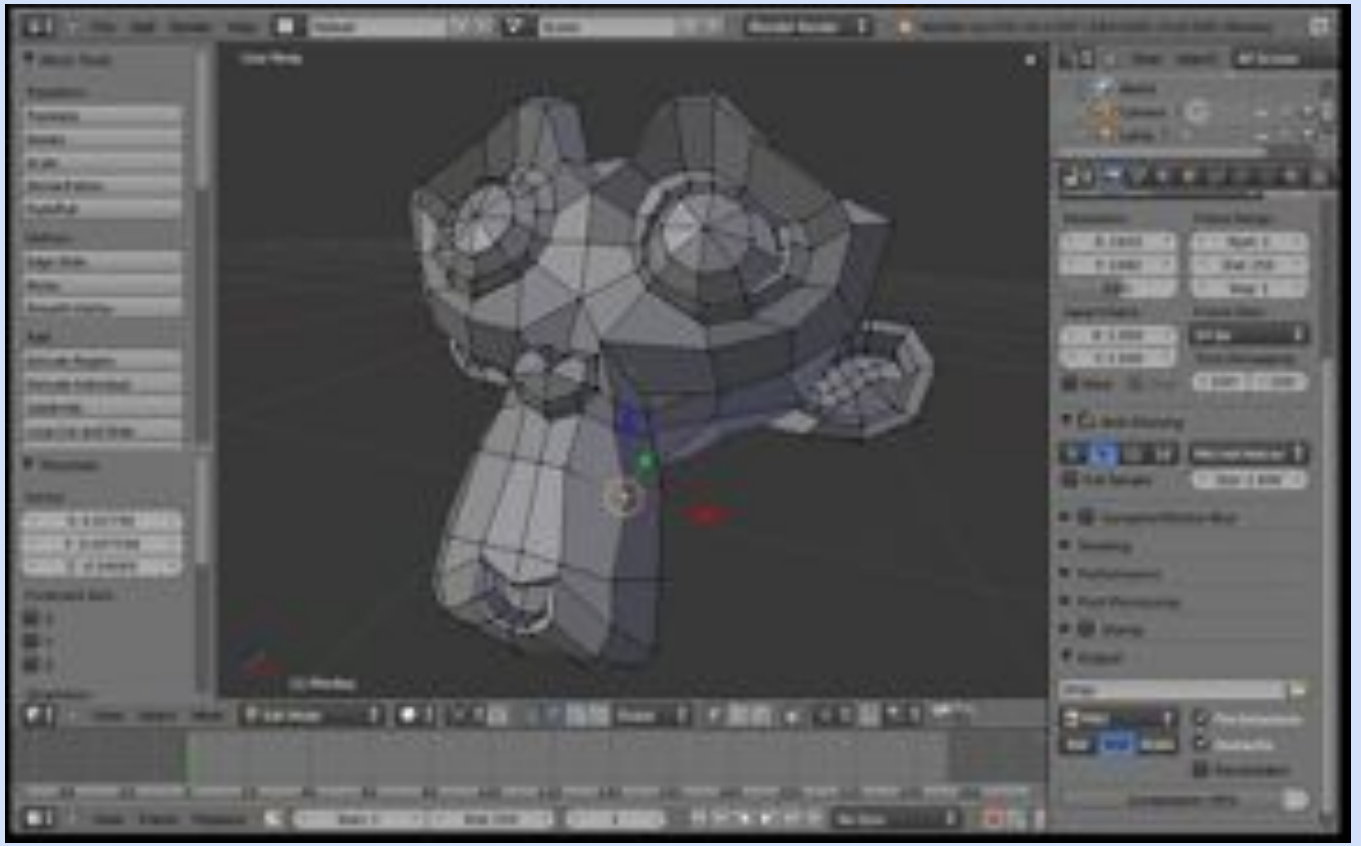
Concept #10: Programming APIs and toolkits for graphics.

There are programming toolkits (and some like Unity are much more) that let hackers write code that generates graphics or manipulates models that are turned into graphics.

```
/* Vertex shader */
uniform float waveTime;
uniform float waveWidth;
uniform float waveHeight;

void main(void)
{
    vec4 v = vec4(gl_Vertex);

    v.z = sin(waveWidth * v.x + waveTime) * cos(waveWidth * v.y + waveTime) * waveHeight;

    gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

There are **low-level APIs** like OpenGL that give you complete control over the graphics system. But this comes at a price: you have to manage memory, rendering pipelines, and all that explicitly. I've been there: it is *fun*, *powerful*, and *awesome, but I am also* ***covered in scars***. Tread lightly!

High-level tools like Unity and **Blender** give you a programmer-friendly API for working with high-level objects. While OpenGL has you manipulating points or (at best) display list structures, Blender has you manipulating Python objects, and you can 'program' these things interactively with a GUI. You can of course drop down and do lower level stuff in Python.

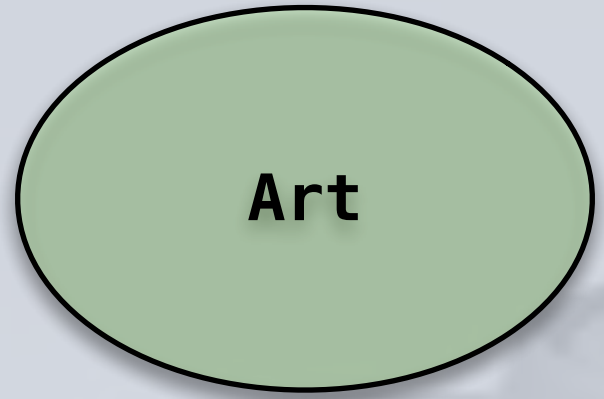


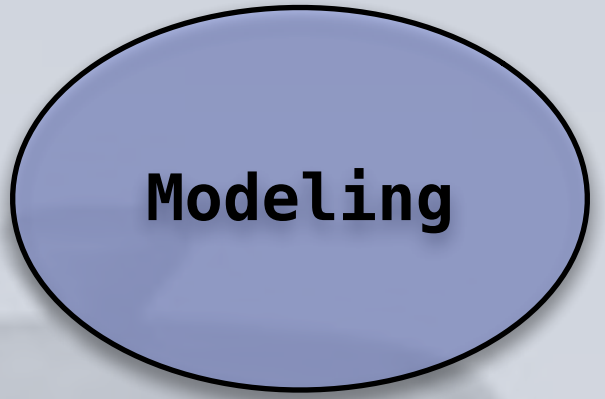
So:
—low-level tools are math- and algorithm heavy.

—high-level tools are Object-oriented and (mostly) graphical, but you can program them with a proper coding language

Computer Graphics Concept Summary!

Concept #1: Pixels are the bridge between computers and people.
Concept #2: We model things to draw using data structures.
Concept #3: Colors are numbers: we can do math with those.
Concept #4: Directions, points can be modeled as vectors (=math).
Concept #5: Matrices can store/compute *transformations*.
Concept #6: Math done (1) in parallel (2) with matrices/vectors.
Concept #7: CG models involve *vectors*, *meshes*, and *textures*.
Concept #8: Can design *with* and *for* computer graphics.
Concept #9: Be clever about how to make CG models.
Concept #10: Use/write APIs and toolkits for graphics.





Many areas in computer graphics to choose from. I hope this has given you a decent overview and has helped make your map of the area a bit less fuzzy!