

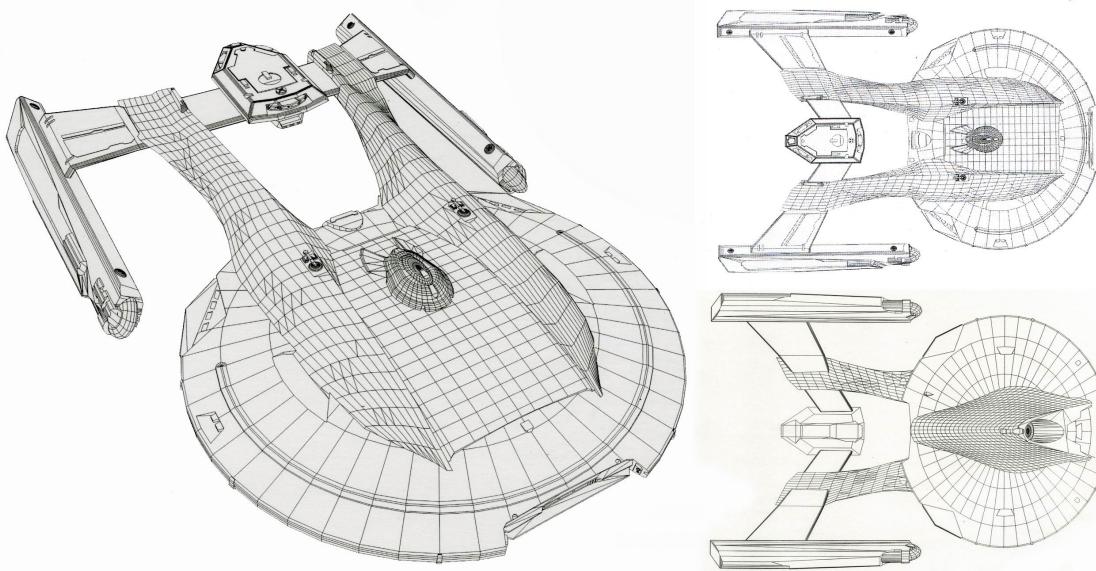
Computer Graphics

Today we'll study the mathematics used to create and manipulate computer graphics.

The computer graphics seen in movies and videogames works in three stages:

1. A 3D model of the scene objects is created;
2. The model is converted into (many small) polygons in 3D that approximate the surfaces of the model; and
3. The polygons are transformed via a linear transformation to yield a 2D representation that can be shown on a flat screen.

There is interesting mathematics in each stage, but the transformations that take place in the third stage are **linear**, and that's what we'll study today.



Initially, object models may be expressed in terms of smooth functions like polynomials.

However the first step is to convert those smooth functions into a set of discrete pieces – coordinates and line segments.

All subsequent processing is done in terms of the discrete coordinates that approximate the shape of the original model.

The reason for this conversion is that most transformations needed in graphics are *linear*.

Expressing the scene in terms of coordinates is equivalent to expressing it in terms of vectors, eg, in \mathbb{R}^3 .

And linear transformations on vectors are always matrix multiplications, so implementation is simple and uniform.

The resulting representation consists of lists of 3D coordinates called *faces*. Each face is a polygon.

The lines drawn between coordinates are implied by the way that coordinates are grouped into faces.

Example.

Here is a view of a ball-like object. It is centered at the origin.

The ball is represented in terms of 3D coordinates, but we are only plotting the *x* and *y* coordinates here.

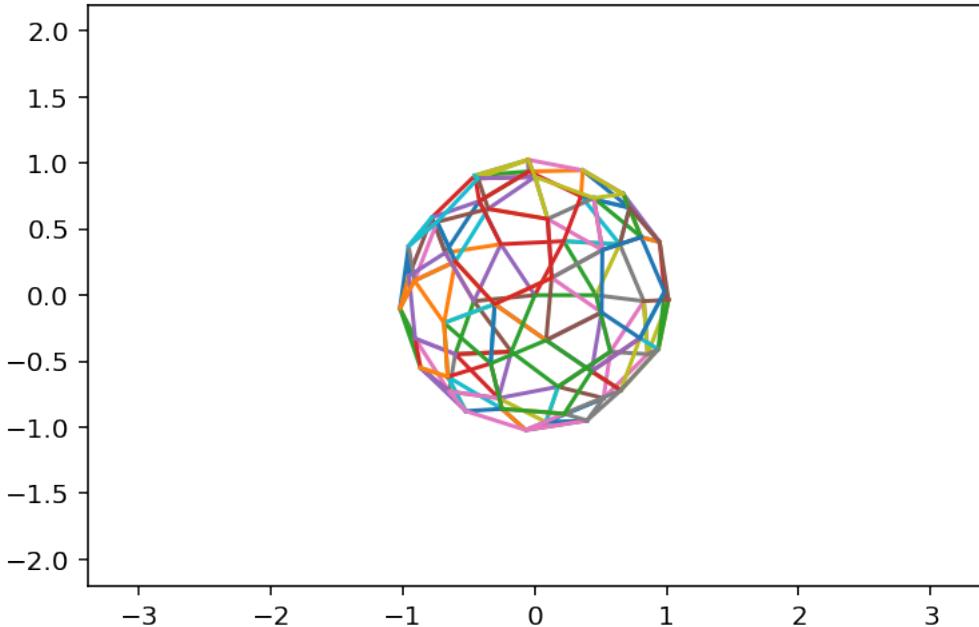
Colors correspond to faces.

```
[7]: # load ball wireframe
import obj2clist as obj
reload(obj)
fp = open('snub_icosidodecahedron.wrl', 'r')
```

```

ball = obj.wrl2flist(fp)
fp.close()
# set up view
fig = plt.figure()
ax = plt.axes(xlim=(-5,5),ylim=(-5,5))
plt.plot(-2,-2,'')
plt.plot(2,2,'')
plt.axis('equal')
for b in ball:
    ax.plot(b[0],b[1])

```



Imagine that we want to circle the camera around the ball. This is equivalent to rotating the ball around the y axis.

This is a linear transformation. We can implement it by multiplying the coordinates of the ball by a *rotation matrix*. To define the rotation matrix, we need to think about what happens to each of the columns of I : $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$.

To rotate through an angle of α radians around the y axis, the vector $\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ goes to $\begin{bmatrix} \cos \alpha \\ 0 \\ \sin \alpha \end{bmatrix}$.

Of course, \mathbf{e}_2 is unchanged.

And $\mathbf{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ goes to $\begin{bmatrix} -\sin \alpha \\ 0 \\ \cos \alpha \end{bmatrix}$.

So the entire rotation matrix is:

$$\begin{bmatrix} \cos \alpha & 0 & -\sin \alpha \\ 0 & 1 & 0 \\ \sin \alpha & 0 & \cos \alpha \end{bmatrix}.$$

```
[5]: sl.hide_code_in_slideshow()
import base64
```

```

# python notebook does not support matplotlib animations
# so these functions create an .mp4 video and then display
# it using inline HTML

# updated to python 3, uses base64 for encoding

# Source: http://nbviewer.ipython.org/url/jakevdp.github.io/downloads/notebooks/AnimationEmbedding
from tempfile import NamedTemporaryFile

VIDEO_TAG = """<video controls>
    <source src="data:video/x-m4v;base64,{0}" type="video/mp4">
    Your browser does not support the video tag.
</video>"""

def anim_to_html(anim):
    if not hasattr(anim, '_encoded_video'):
        with NamedTemporaryFile(suffix='.mp4') as f:
            anim.save(f.name, fps=20, extra_args=['-vcodec', 'libx264'])
            video = open(f.name, "rb").read()
        anim._encoded_video = base64.b64encode(video).decode('utf-8')

    return VIDEO_TAG.format(anim._encoded_video)

def display_animation(anim):
    plt.close(anim._fig)
    return HTML(anim_to_html(anim))

[6]: # set up view
import matplotlib.animation as animation
fig = plt.figure()
ax = plt.axes(xlim=(-5,5),ylim=(-5,5))
plt.plot(-2,-2,'')
plt.plot(2,2,'')
plt.axis('equal')
ballLines = []
for b in ball:
    ballLines += ax.plot([],[])
def animate(i):
    angle = 2.0 * np.pi * (i/100.0)
    rotationMatrix = np.array([[np.cos(angle),0,-np.sin(angle)],
                               [0,1,0],[np.sin(angle),0,np.cos(angle)]]])
    for b,l in zip(ball,ballLines):
        rb = rotationMatrix.dot(b)
        l.set_data(rb[0],rb[1])
    fig.canvas.draw()
anim = animation.FuncAnimation(fig,animate,frames=100,interval=100,repeat=False,blit=False)
# this function requires ffmpeg to be installed on your system
display_animation(anim)

```

MovieWriter ffmpeg unavailable.

```

KeyError                                         Traceback (most recent call last)

~/anaconda3/lib/python3.5/site-packages/PIL/Image.py in save(self, fp, format, **params)
1929         try:
-> 1930             format = EXTENSION[ext]
1931         except KeyError:

KeyError: '.mp4'

During handling of the above exception, another exception occurred:

ValueError                                         Traceback (most recent call last)

<ipython-input-6-fae05a6d7db6> in <module>()
19 anim = animation.FuncAnimation(fig,animate,frames=100,interval=100,repeat=False,blit=False)
20 # this function requires ffmpeg to be installed on your system
---> 21 display_animation(anim)

<ipython-input-5-8fe9c36ed38d> in display_animation(anim)
27 def display_animation(anim):
28     plt.close(anim._fig)
---> 29     return HTML(anim_to_html(anim))

<ipython-input-5-8fe9c36ed38d> in anim_to_html(anim)
19     if not hasattr(anim, '_encoded_video'):
20         with NamedTemporaryFile(suffix='.mp4') as f:
---> 21             anim.save(f.name, fps=20, extra_args=['-vcodec', 'libx264'])
22             video = open(f.name, "rb").read()
23             anim._encoded_video = base64.b64encode(video).decode('utf-8')

~/anaconda3/lib/python3.5/site-packages/matplotlib/animation.py in save(self, filename, writer,
1198                                         # TODO: See if turning off blit is really necessary
1199                                         anim._draw_next_frame(d, blit=False)
-> 1200                                         writer.grab_frame(**savefig_kwargs)
1201
1202                                         # Reconnect signal for first draw if necessary

~/anaconda3/lib/python3.5/contextlib.py in __exit__(self, type, value, traceback)
64         if type is None:
65             try:
---> 66                 next(self.gen)
67             except StopIteration:
68                 return False

~/anaconda3/lib/python3.5/site-packages/matplotlib/animation.py in saving(self, fig, outfile, dp

```

```

239             yield self
240         finally:
--> 241             self.finish()
242
243

~/anaconda3/lib/python3.5/site-packages/matplotlib/animation.py in finish(self)
610     self._frames[0].save(
611         self._outfile, save_all=True, append_images=self._frames[1:],
--> 612         duration=int(1000 / self.fps))
613
614

~/anaconda3/lib/python3.5/site-packages/PIL/Image.py in save(self, fp, format, **params)
1930         format = EXTENSION[ext]
1931     except KeyError:
-> 1932         raise ValueError('unknown file extension: {}'.format(ext))
1933
1934     if format.upper() not in SAVE:

ValueError: unknown file extension: .mp4

```

Translation

Manipulating graphics objects using matrix multiplication is very convenient.

However, there is a common operation that is **not** a linear transformation: *translation*, aka motion.

The standard way to avoid this difficulty is to use what are called **homogeneous coordinates**.

We identify each point $\begin{bmatrix} x \\ y \\ z \end{bmatrix} \in \mathbb{R}^3$ with a corresponding point $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \in \mathbb{R}^4$.

The coordinates of the point in \mathbb{R}^4 are the homogeneous coordinates for the point in \mathbb{R}^3 .

The extra component gives us a constant that we can scale and add to the other coordinates, as needed, via matrix multiplication.

This means for 3D graphics, all transformation matrices are 4×4 .

Example.

Let's say we want to move a point (x, y, z) to location $(x + h, y + k, z + m)$.

We represent the point in homogeneous coordinates as $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$.

The transformation corresponding to this 'translation' is:

$$\begin{bmatrix} 1 & 0 & 0 & h \\ 0 & 1 & 0 & k \\ 0 & 0 & 1 & m \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + h \\ y + k \\ z + m \\ 1 \end{bmatrix}.$$

If we only consider x, y , and z this is not a linear transformation. But of course, in \mathbb{R}^4 this most definitely is a linear transformation.

We have ‘sheared’ the fourth dimension, which affects the other three. A very useful trick!

Constructing Matrices for Homogeneous Coordinates

For any transformation A that is linear in \mathbb{R}^3 (such as scaling, rotation, reflection, shearing, etc.), we can construct the corresponding matrix for homogeneous coordinates quite simply:

If

$$A = \begin{bmatrix} \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare \end{bmatrix}$$

Then the corresponding transformation for homogeneous coordinates is:

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

In other words, when performing a linear transformation on x, y , and z , one simply ‘carries along’ the extra coordinate without modifying it.

Perspective Projections

There is another nonlinear transformation that is important in computer graphics: perspective.

Happily, we will see that homogeneous coordinates allow us to capture this too as a linear transformation in \mathbb{R}^4 .

The eye, or a camera, captures light (essentially) in a single location, and hence gathers light rays that are converging.

So, to portray a scene with realistic appearance, it is necessary to reproduce this effect.

The effect can be thought of as “nearer objects are larger than further objects.”

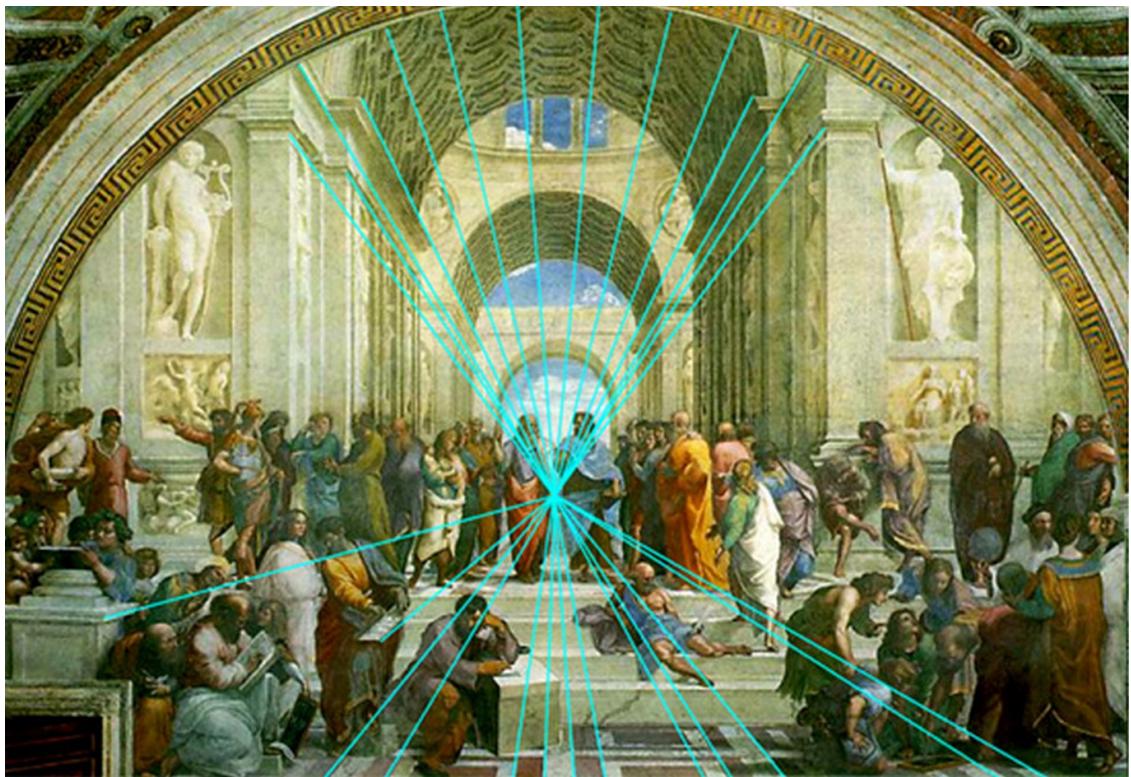
This was (re)discovered by Renaissance artists (supposedly first by Filippo Brunelleschi, around 1405).

Here is a famous example: Raphael’s *School of Athens* (1510).

```
[8]: sl.hide_code_in_slideshow()
#source: http://www.webexhibits.org/sciartperspective/tylerperspective.html
display(Image("images/Raphael-SchoolofAthens.jpg", width=750))
```



```
[9]: sl.hide_code_in_slideshow()  
display(Image("images/Raphael-SchoolofAthens-Perspective.jpg", width=750))
```



We now understand that this effect interacts in a powerful way with neural circuitry in our brains.
The mind reconstructs a sense of three dimensions from the two dimensional information presented to it by the retina.

This is done by sophisticated processing in the visual cortex, which is a fairly large portion of the brain.

```
[10]: sl.hide_code_in_slideshow()
fig = plt.figure()
ax = plt.axes(xlim=(-5,5),ylim=(-5,5))
plt.plot(-2,-2,'')
plt.plot(2,2,'')
plt.axis('equal')
fp = open('cube.obj','r')
cube = obj.obj2flist(fp)
fp.close()
cube = obj.homogenize(cube)
cubeLines = []
for c in cube:
    cubeLines += ax.plot([],[])
def animate(i):
    angle = 2.0 * np.pi * (i/100.0)
    P = np.array([[1.,0,0,0],[0,1.,0,0],[0,0,0,0],[0,0,-1./8,1]]).dot(np.array([[np.cos(angle),0,0,0],[0,np.sin(angle),0,0],[0,0,1,0],[0,0,0,1]]))
    for b,l in zip(cube,cubeLines):
        rb = P.dot(b)
        l.set_data(rb[0]/rb[3],rb[1]/rb[3])
    fig.canvas.draw()
anim = animation.FuncAnimation(fig,animate,frames=101,interval=100,repeat=False,blit=False)
# this function requires ffmpeg to be installed on your system
display_animation(anim)
```

MovieWriter ffmpeg unavailable.

KeyError Traceback (most recent call last)

```
~/anaconda3/lib/python3.5/site-packages/PIL/Image.py in save(self, fp, format, **params)
1929         try:
-> 1930             format = EXTENSION[ext]
1931         except KeyError:
```

KeyError: '.mp4'

During handling of the above exception, another exception occurred:

ValueError Traceback (most recent call last)

```
<ipython-input-10-e6cd327027d1> in <module>()
21 anim = animation.FuncAnimation(fig,animate,frames=101,interval=100,repeat=False,blit=False)
```

```

22 # this function requires ffmpeg to be installed on your system
---> 23 display_animation(anim)

<ipython-input-5-8fe9c36ed38d> in display_animation(anim)
27 def display_animation(anim):
28     plt.close(anim._fig)
---> 29     return HTML(anim_to_html(anim))

<ipython-input-5-8fe9c36ed38d> in anim_to_html(anim)
19     if not hasattr(anim, '_encoded_video'):
20         with NamedTemporaryFile(suffix='.mp4') as f:
---> 21             anim.save(f.name, fps=20, extra_args=['-vcodec', 'libx264'])
22             video = open(f.name, "rb").read()
23             anim._encoded_video = base64.b64encode(video).decode('utf-8')

~/anaconda3/lib/python3.5/site-packages/matplotlib/animation.py in save(self, filename, writer,
1198                                         # TODO: See if turning off blit is really necessary
1199                                         anim._draw_next_frame(d, blit=False)
-> 1200                                         writer.grab_frame(**savefig_kwargs)
1201
1202     # Reconnect signal for first draw if necessary

~/anaconda3/lib/python3.5/contextlib.py in __exit__(self, type, value, traceback)
64         if type is None:
65             try:
---> 66                 next(self.gen)
67             except StopIteration:
68                 return False

~/anaconda3/lib/python3.5/site-packages/matplotlib/animation.py in saving(self, fig, outfile, dp
239         yield self
240     finally:
--> 241         self.finish()
242
243

~/anaconda3/lib/python3.5/site-packages/matplotlib/animation.py in finish(self)
610         self._frames[0].save(
611             self._outfile, save_all=True, append_images=self._frames[1:],
--> 612             duration=int(1000 / self.fps))
613
614

~/anaconda3/lib/python3.5/site-packages/PIL/Image.py in save(self, fp, format, **params)
1930             format = EXTENSION[ext]
1931         except KeyError:
--> 1932             raise ValueError('unknown file extension: {}'.format(ext))

```

```

1933
1934     if format.upper() not in SAVE:

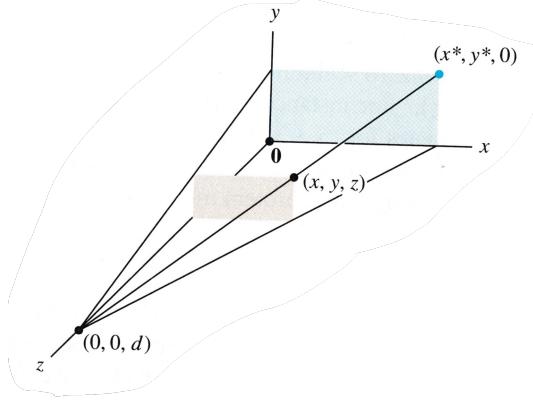
```

```
ValueError: unknown file extension: .mp4
```

Notice that when the image is stationary, it appears flat (like a picture frame). As soon as it starts to move, it springs into 3D in perception.

Computing Perspective.

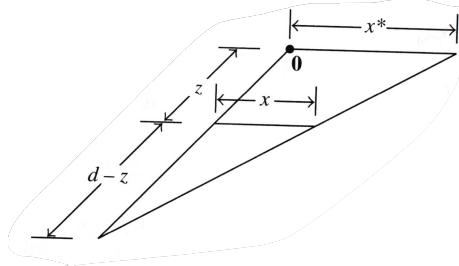
A standard setup for computing a perspective transformation is as shown in this figure:



For simplicity, we will let the xy -plane represent the screen. This is called the *viewing plane*.

The eye/camera is located on the z axis, at the point $(0, 0, d)$. This is call the *center of projection*.

A *perspective projection* maps each point (x, y, z) onto an image point $(x^*, y^*, 0)$ so that the center of projection and the two points are all on the same line.



The way to compute the projection is using similar triangles.

The triangle in the xz -plane shows the lengths of corresponding line segments.

Similar triangles show that

$$\frac{x^*}{d} = \frac{x}{d - z}$$

and so

$$x^* = \frac{dx}{d - z} = \frac{x}{1 - z/d}.$$

Notice that the function $T : (x, z) \mapsto \frac{x}{1 - z/d}$ is **not** linear.

Using homogeneous coordinates, we can construct a linear version of T in \mathbb{R}^4 .

To do so, we establish the following convention: we will allow the fourth coordinate to vary away from

1.

However, when we plot, for a point $\begin{bmatrix} x \\ y \\ z \\ h \end{bmatrix}$ we will plot the point $\begin{bmatrix} \frac{x}{h} \\ \frac{y}{h} \\ \frac{z}{h} \\ \bar{h} \end{bmatrix}$.

In this way, by dividing the x, y, z coordinates by the h coordinate, we can implement a **nonlinear** transform in \mathbb{R}^3 .

So, to implement the perspective transform, we want $\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ to map to $\begin{bmatrix} \frac{x}{1-z/d} \\ \frac{y}{1-z/d} \\ 0 \\ 1 \end{bmatrix}$.

The way we will implement this is to actually cause it to map to $\begin{bmatrix} x \\ y \\ 0 \\ 1-z/d \end{bmatrix}$.

Then, when we plot (dividing x and y by the h value) we will get the proper transform.

The matrix that implements this transformation is quite simple:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 1 \end{bmatrix}.$$

So:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ 1-z/d \end{bmatrix}.$$

Composing Transformations

One big payoff for casting all graphics operations as linear transformations comes in the **composition** of transformations.

Consider two linear transformations T and S . For example, T could be a scaling and S could be a rotation. Assume S is implemented by a matrix A and T is implemented by a matrix B .

To first scale and then rotate a vector \mathbf{x} we would compute $S(T(\mathbf{x}))$.

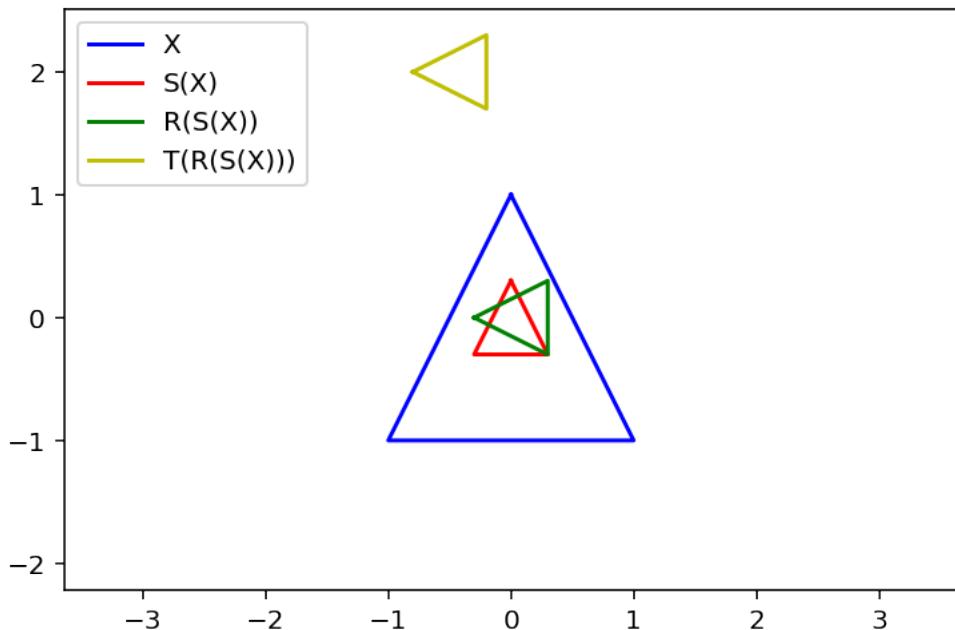
Of course this is implemented as $A(B\mathbf{x})$.

But note that this is the same as $(AB)\mathbf{x}$. In other words, AB is a *single matrix that both scales and rotates \mathbf{x}* .

By extension, we can combine any arbitrary sequence of linear transformations into a single matrix. This greatly simplifies high-speed graphics.

Note though that if $C = AB$, then C is the transformation that *first* applies B , *then* applies A .

Example. Let's work in homogenous coordinates for points in \mathbb{R}^2 . Find the 3×3 matrix that corresponds to the composite transformation of first scaling by 0.3, then rotation of 90° about the origin, and finally a translation of $\begin{bmatrix} -0.5 \\ 2 \end{bmatrix}$ to each point of a figure.



The scaling matrix is:

$$S = \begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 0.3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix is:

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

(Note that $\sin 90^\circ = 1$ and $\cos 90^\circ = 0$.)

The translation matrix is:

$$T = \begin{bmatrix} 1 & 0 & -0.5 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

So the matrix for the composite transformation is:

$$TRS = \begin{bmatrix} 1 & 0 & -0.5 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.3 & 0.3 & 0 \\ 0 & 0.3 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -0.3 & -0.5 \\ 0.3 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix}.$$

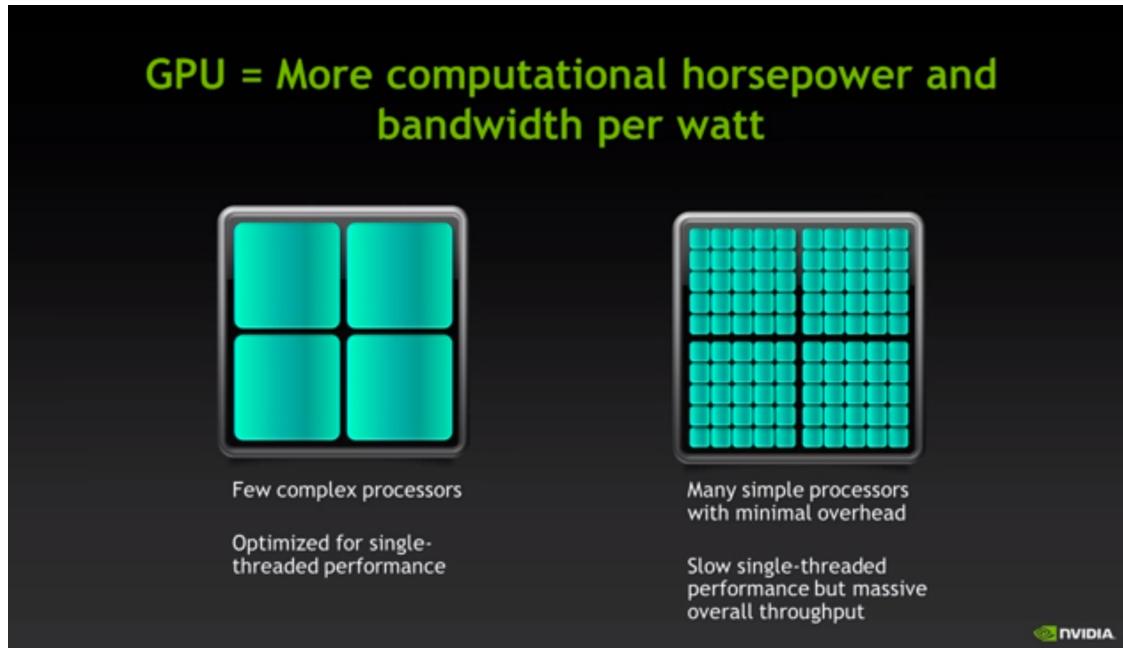
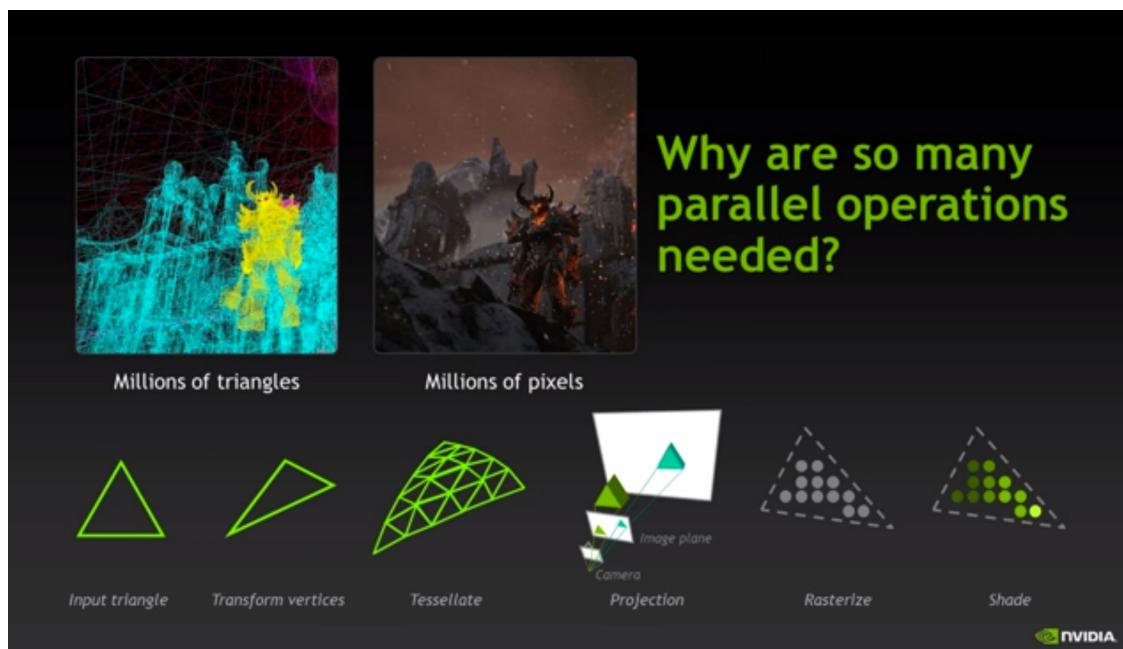
Homework 7

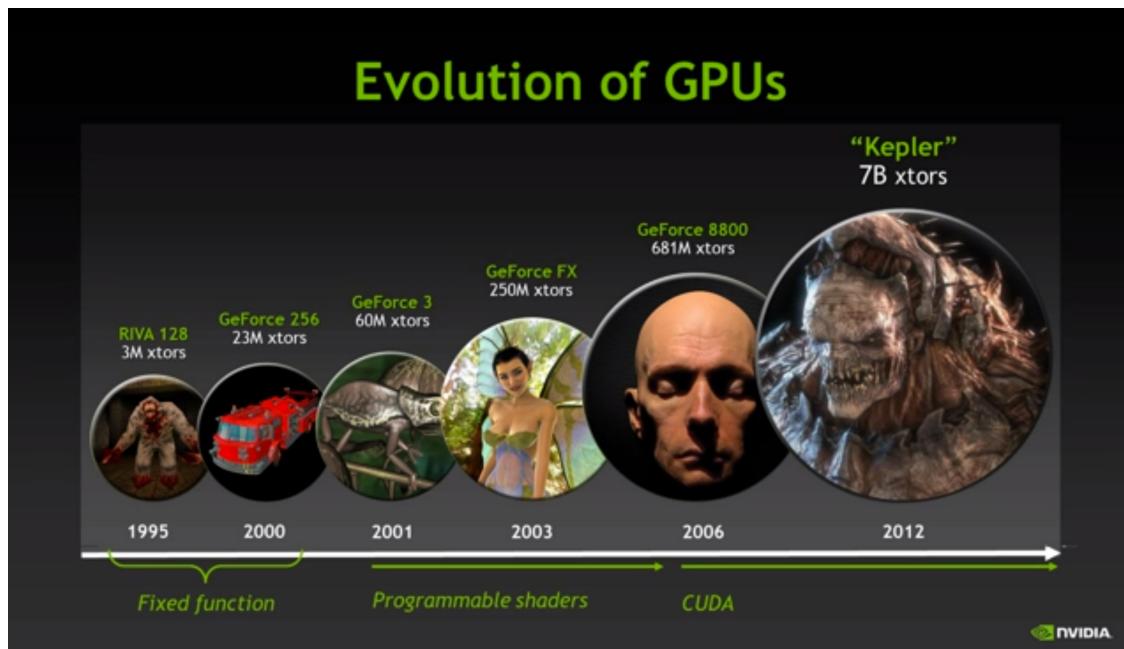
Homework 7 will allow you to explore these ideas by computing various transformation matrices. I will give you some 3D wireframe objects and you will compute the necessary matrices to create a simple animation.

Here is what your finished product will look like:

<IPython.core.display.HTML object>

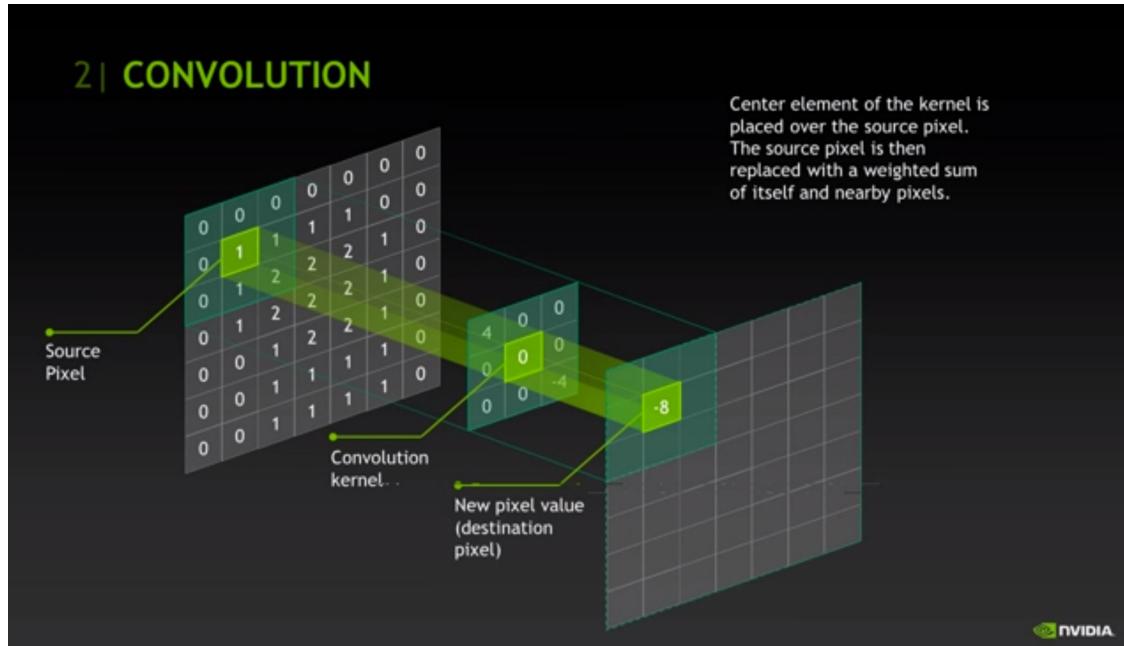
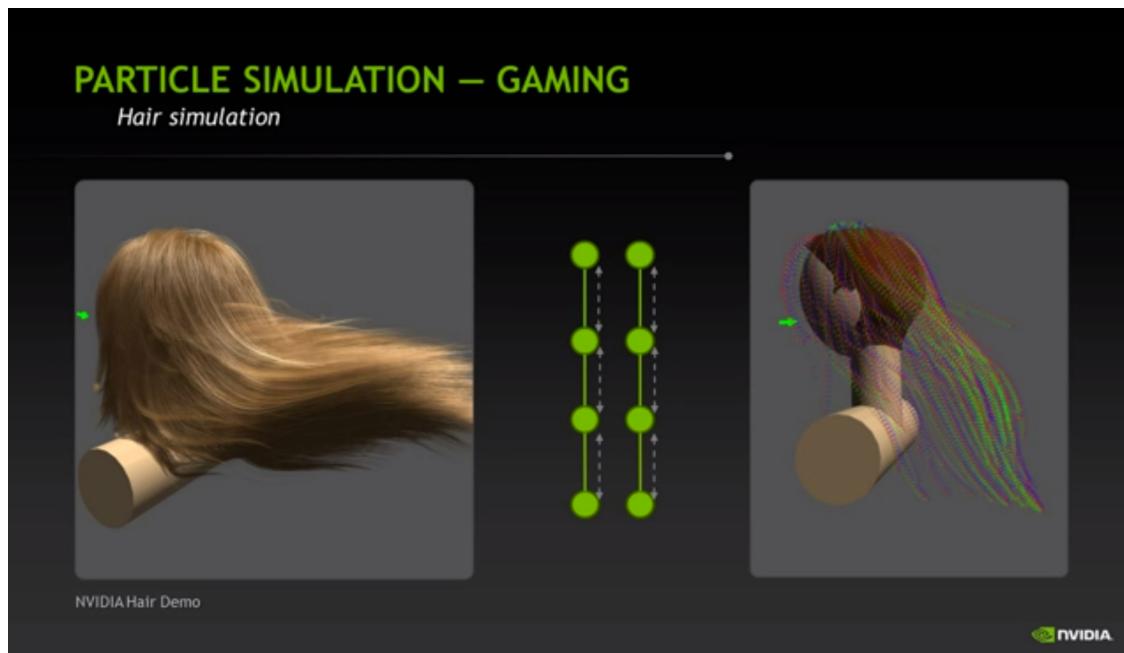
Fast Computation of Linear Systems using Graphics Processing Units (GPUs)





Examples of Other Linear Systems Used in Computer Graphics





CONVOLUTION – GAMING

Depth of field



Halo 3® Bungie Studios

