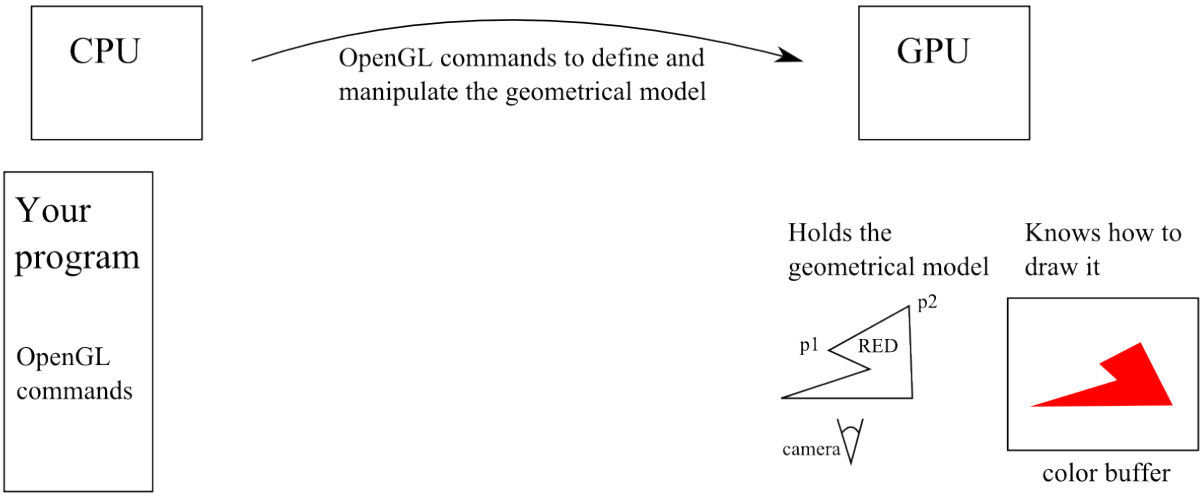
SE313: Computer Graphics and Visual Programming

Computer Graphics Notes – Gazihan Alankus, Fall 2011

Computer Graphics

Geometrical model on the computer -> Image on the screen

How it works in your computer



In the past GPU was a separate computer accessed over the network.

Today it's still like a separate computer, but in your graphics card.

Using OpenGL commands you tell it how things are and how it should transform things, it does it for you and draws it.

Points, vectors, transformations, etc

Check out these excellent slides and notes.

They help you define your geometric model conceptually.

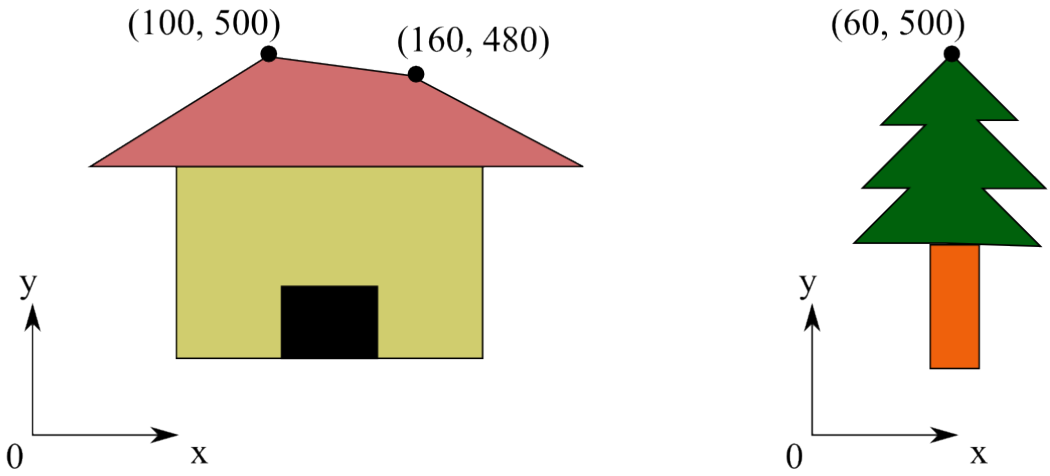
Different coordinate frames

"Here is the center of the world" –Nasreddin Hoca

You can take your reference point to be anywhere you want, as long as you are consistent. We'll see that different things have different reference frames.

Modeling coordinates

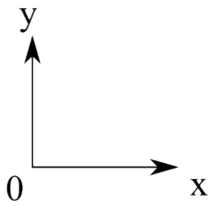
An artist creates a model and gives it to you. It inherently has a coordinate frame.



Let's say the artist creates a house and a tree in two different files. Both files have an origin and coordinate frames. These coordinates of the house and the tree are called **modeling coordinates**.

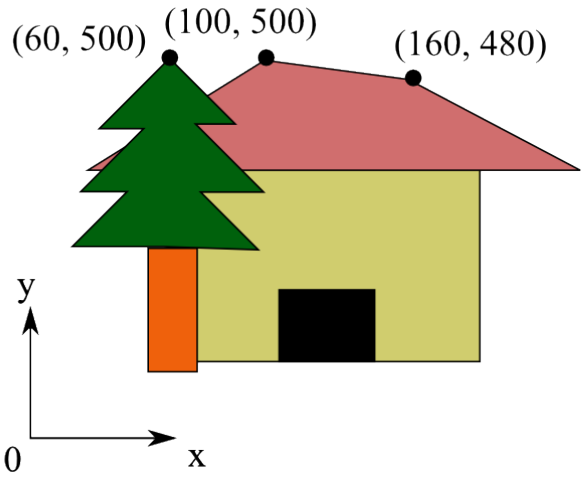
World coordinates

The art assets by themselves are not very useful and we want to create a scene out of them. Therefore, we want to bring them in to a new coordinate frame. This is called **world coordinates** or **scene coordinates**.

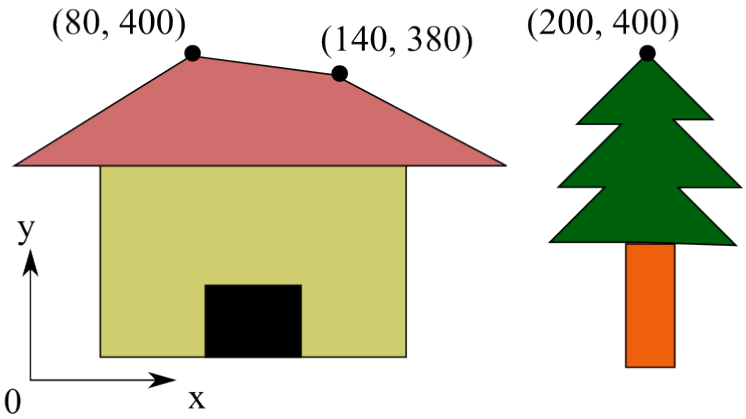


Above is an empty world with a coordinate frame.

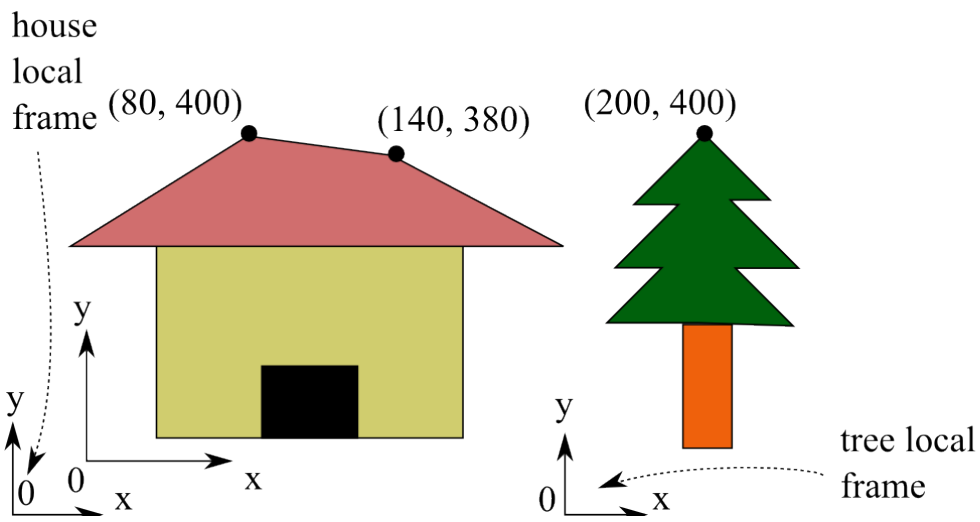
Now we want to place our house and the tree in the world. If we just put the geometric models in the world, here is what it would look like:



This is not what we want. We want the house to be lower and the tree on the right. We add and subtract to the house's and the tree's coordinates to get them to where we want them:



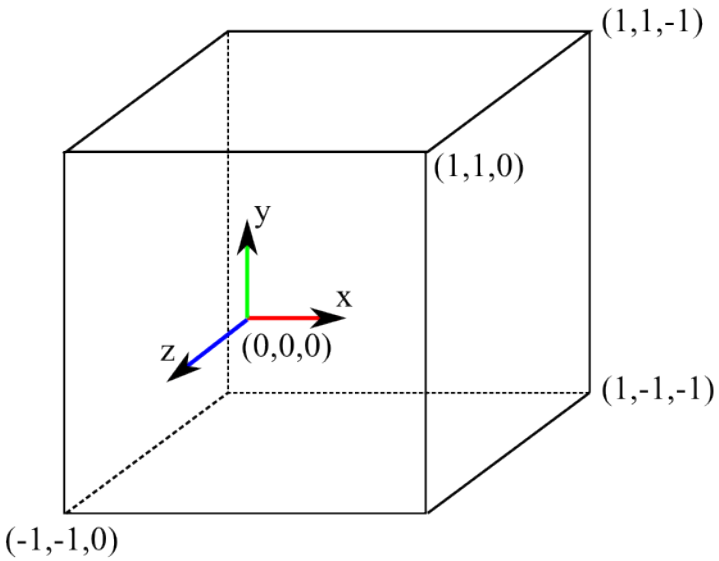
Even though we modified locations of the vertices of the tree and the house, we brought them to a common coordinate frame, which is the world coordinate frame. We can still keep track of the previous coordinate frames of the house and the tree in this new coordinates:



These are called “local coordinate frames”. Everything in the scene has a local coordinate frame, and everything is transformed to find their places in the world coordinate frame.

Viewing coordinates

Bringing objects to their places in the world is useful, but we need a camera to create an image of the scene. Our camera can be located in the scene similar to the other objects. The camera also has a coordinate frame. There is a box that the camera can see.



This is the viewing volume in the camera’s coordinates. Everything in this volume gets orthogonally projected into 2D. Imagine pushing on the box along the z direction to make it flat, that’s exactly how it turns into 2D.

When you look at an OpenGL window, this viewing volume is what you are looking at. Therefore, to see objects in your OpenGL window, you have to bring your world into this viewing volume. You do that by transforming your world to the viewing volume. The final coordinates are called **viewing coordinates**.

OpenGL

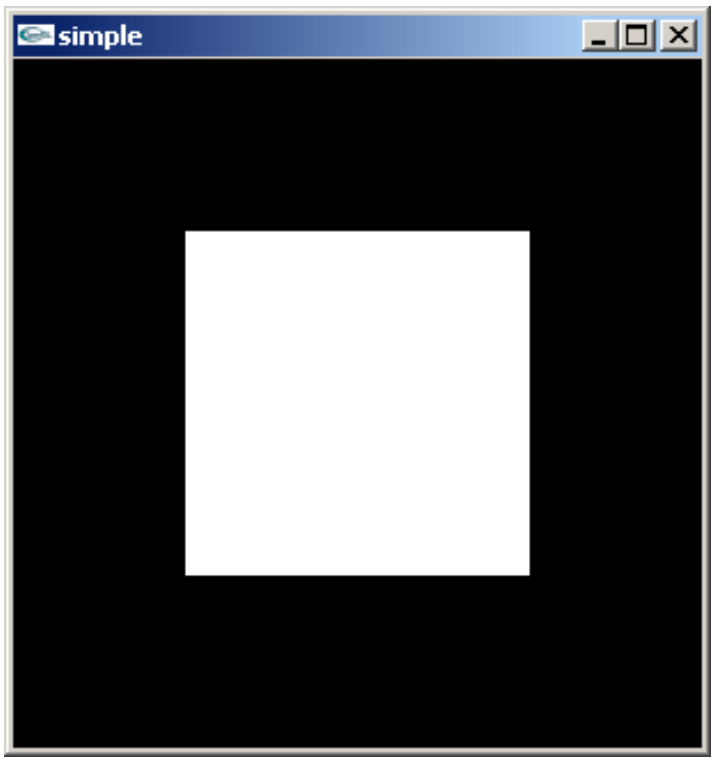
OpenGL is a state machine. You give it commands to change its state, and you tell it to draw things using the current state.

Introduction

You can draw primitive geometric shapes in OpenGL by giving vertices (points) one by one to OpenGL. Ideally an artist would create it in an authoring software (like the tree and the house above). But we will do it by hand for now. Example:

```
glBegin(GL_POLYGON);
      glVertex2f(-0.5, -0.5);
      glVertex2f(-0.5, 0.5);
      glVertex2f(0.5, 0.5);
      glVertex2f(0.5, -0.5);
glEnd();
```

This tells OpenGL to start drawing a polygon, gives it four 2D points (z is assumed 0 then) and ends the polygon. If we do not do anything special, this gets drawn in the viewing coordinates (see figure above). Since the corners of the viewing volume are 1, this set of points will draw a square that takes ¼ of the screen and will be in the middle.



This is the output that confirms what we said. You need glut, or some other windowing tool to create the window for you and take care of other things as well. Below is a complete program:

```
#include <GL/glut.h> //you may have to remove GL/ in windows
void mydisplay(){
	glClear(GL_COLOR_BUFFER_BIT);
	glBegin(GL_POLYGON);
		glVertex2f(-0.5, -0.5);
		glVertex2f(-0.5, 0.5);
		glVertex2f(0.5, 0.5);
		glVertex2f(0.5, -0.5);
	glEnd();
	glFlush();
}
int main(int argc, char** argv){
	glutCreateWindow("simple");
```

```
    glutDisplayFunc(mydisplay);

    glutMainLoop();

}
```

Glut

Glut opens the window for you and runs some functions that you give to it. These are called “callback functions” because glut calls them back when it needs to.

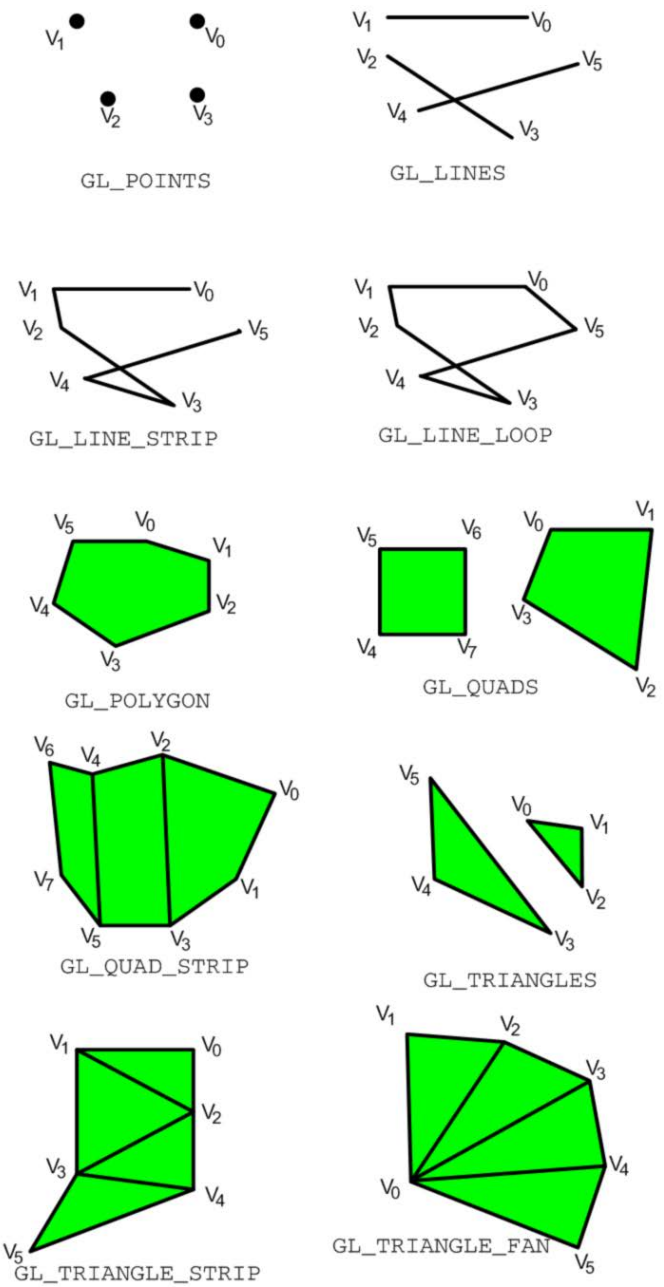
```
glutDisplayFunc(renderScene);

glutReshapeFunc(changeSize);

glutIdleFunc(renderScene);

glutKeyboardFunc(processNormalKeys);

glutSpecialFunc(processSpecialKeys);
```

You create the functions that we gave as arguments (renderScene, changeSize, etc.). Glut calls them when necessary. Glut takes care of keyboard input, too.

There is a great tutorial here

Primitives

In addition to polygons, we can also draw the following primitives using vertices:



Attributes of Primitives

OpenGL is stateful. You can set the state of OpenGL to determine how it will draw things. Some basic attributes are:

Color

glColor3f(1.0, 0.0, 0.0)

Point size

glPointSize(2.0)

Line width
glLineWidth(2.0)

Line pattern
glEnable(GL_LINE_STIPPLE)

glLineStipple(1, 0x1C47)

glDisable(GL_LINE_STIPPLE)

Color blending
glEnable(GL_BLEND);

glDisable(GL_BLEND);

Polygon modes
glPolygonMode()

Play with them to see how they change things. Google them and read man pages.

OpenGL and Transformations
As we said, OpenGL is a state machine. Things like color are part of state. OpenGL also keeps track of the following transformation matrices:

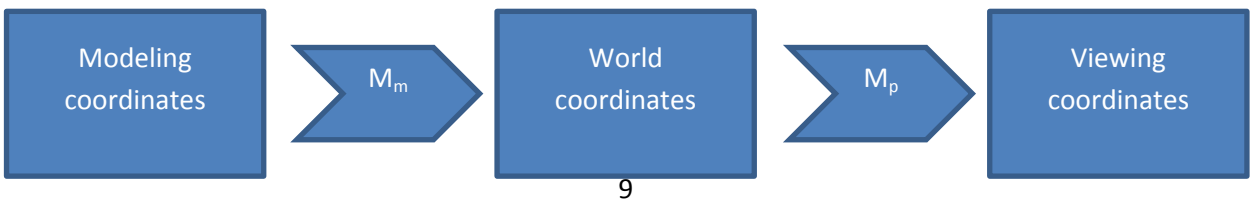
- Projection matrix (M_p)
    - glMatrixMode(GL_PROJECTION)
- Modelview matrix (M_m)
    - glMatrixMode(GL_MODELVIEW)

These are two matrices that OpenGL keeps track of. It uses them whenever you draw something.

The modelview matrix is the transformation that brings your models from the modeling coordinates to world coordinates.

The projection matrix is the transformation that brings your world to the viewing coordinates. By default this is orthogonal. A popular alternative is perspective projection, which distorts the space to make far things smaller.

Whenever you give a point p to OpenGL and tell it to draw it, OpenGL brings it to the viewing volume using the following transformation: $p_v = M_p M_m p$. That is, first it transforms the point with the modelview matrix (M_m) to bring it to where it should be in your world. Then, it transforms it with the projection matrix (M_p) to bring it to the viewing volume and then draws it there.



OpenGL makes this distinction so that you don't have to worry about the camera when you are dealing with your world. This enables you to simply change the projection matrix to move the camera around, or move the modelview matrix to move an object around. To modify one of these matrices, you select it first with glMatrixMode(GL_PROJECTION) or glMatrixMode(GL_MODELVIEW). You only do this when you want to modify that matrix.

If you want, you can multiply these matrices directly with new matrices:

```
glMatrixMode(GL_MODELVIEW);

glLoadIdentity();

glMultMatrixf(m); //this is an array with 16 matrix elements

glRotate(45, 1.0, 0.0, 0.0); //this multiplies the current matrix with
a rotation matrix, around x and 45 degrees

glTranslate(1.0, 0.0, 2.0); //this multiplies the current matrix with
a translation matrix
```

When you do this, it gets the current matrix M0, multiplies it with the matrix M, finds the new matrix M1, and puts it as the new matrix. Here is the exact multiplication M1 = M0.M

You can think of this as "going inside the model". The leftmost matrix in this series of multiplications transforms everything that comes after it. Every matrix multiplication feels like a "local" transformation with respect to it.

Setting up the projection matrix

You usually set your projection matrix in the beginning once. OpenGL provides special functions to save you from dealing with matrices directly:

```
glMatrixMode(GL_PROJECTION)

glLoadIdentity();

glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);  //the viewing volume

gluPerspective( 45.0, (GLfloat)(width)/(GLfloat)(height), 0.1f, 500.0
); //perspective projection
```

Setting up the modelview matrix

Before you draw something, you can modify the current modelview matrix to determine where in the world what you draw will be at.

```
glMatrixMode(GL_MODELVIEW);
```

```
glLoadIdentity();

glRotate(45, 1.0, 0.0, 0.0); //this multiplies the current matrix with a rotation matrix, around x and 45 degrees

glTranslate(1.0, 0.0, 2.0); //this multiplies the current matrix with a translation matrix
```

This makes your modelview matrix M_r . M_t.

You have to keep track of where is what, and ensure that the appropriate transformation matrix is stored in the modelview matrix.

You can set the matrix with

```
glLoadMatrix(m);
```

and get it with

```
glGet(GL_MODELVIEW_MATRIX, m);

glGet(GL_PROJECTION_MATRIX, m);
```

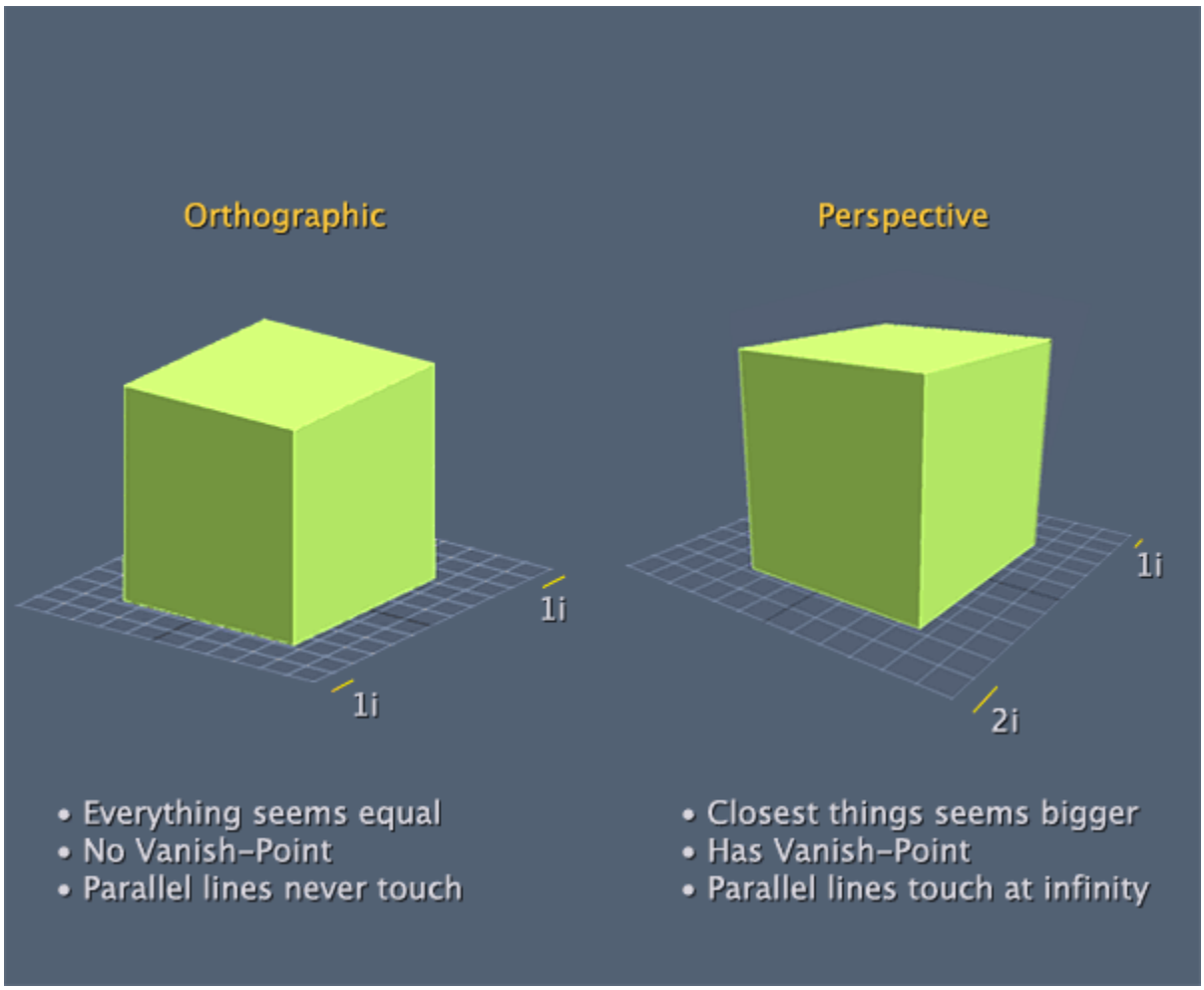
Continued on Week 6:

We mentioned that you can get the current active matrix using **`glGet*()`**. Please take a look at the source code here to see a sample usage of it. Especially check out the **`printTransformationState()`** function and observe that this is exactly how you get the projection matrix out:

```
GLfloat projectionMatrix[16];

glGetFloatv(GL_PROJECTION_MATRIX, projectionMatrix);
```

Orthographic vs. Perspective Projection

Before moving on, let's finish our discussion about perspectives and the camera. OpenGL has two types of projection ability, which is shown in the image below:

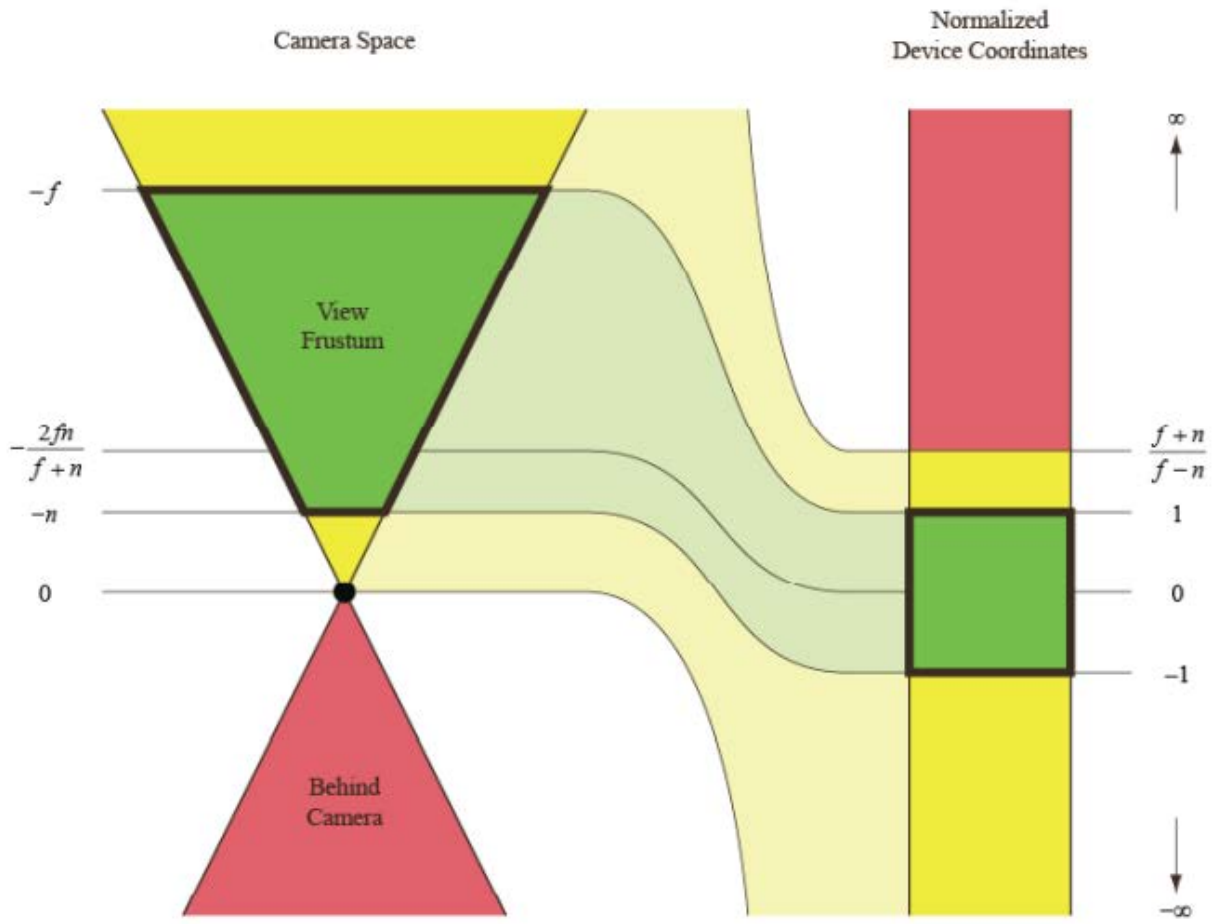


[Images are borrowed from http://db-in.com]

By default the projection is orthographic. In orthographic projection when things move away they do not get smaller. In perspective projection, they become smaller like it is in the real world. Both have their uses.

In orthographic projection, we simply move and scale the world to fit them in our viewing half cube that we talked about before, and project it to 2D.

In perspective projection, we “warp” the space so that a cut pyramid (frustum) in the world is mapped to our viewing halfcube. When it is projected into 2D, we get our image.

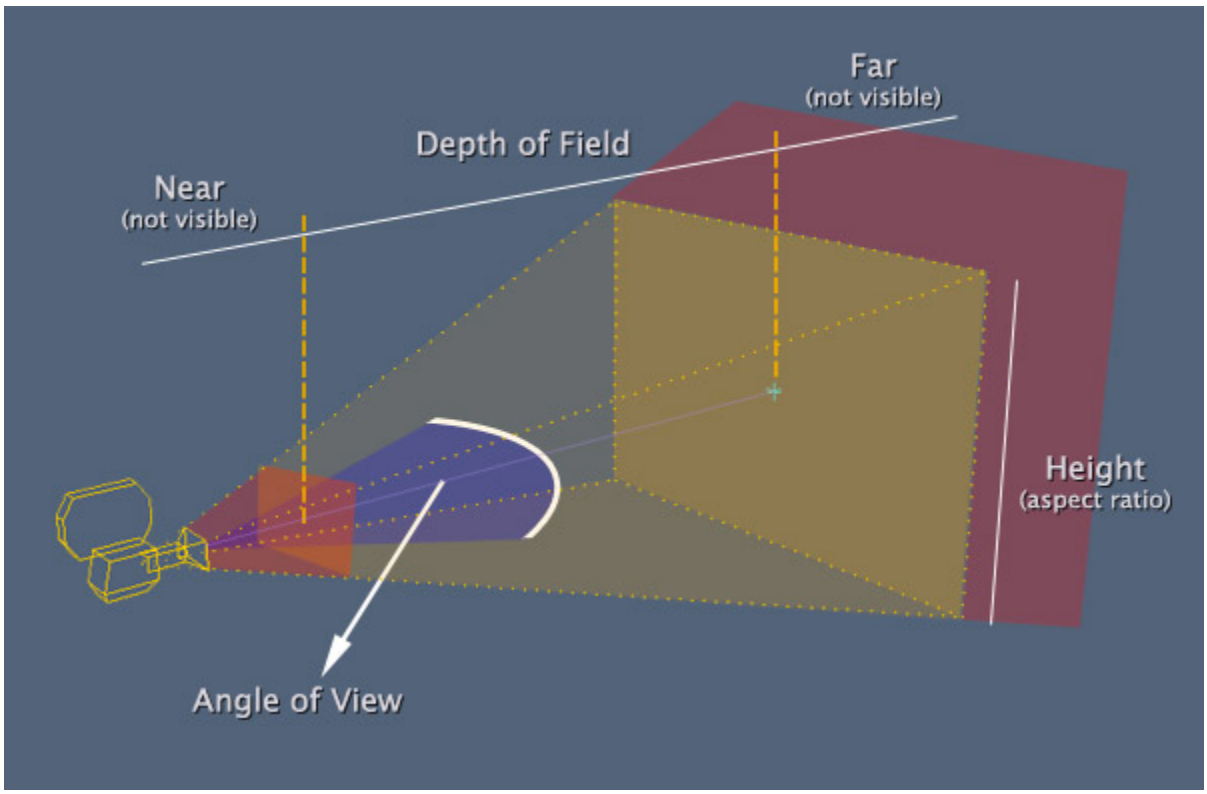


[Image borrowed from http://www.terathon.com/gdc07_lengyel.pdf]

The above image summarizes this “warping” process.

We can set the parameters of the orthographic projection using glOrtho(left, right, bottom, top, near, far), or if we don’t care about the z axis, gluOrtho2D(left, right, bottom, top).

We can set a perspective projection using glFrustum(left, right, bottom, top, near, far) or gluPerspective(fovy angle, aspect ratio, neardistance, fardistance). In both of them, the eye is located at the origin, there is a near plane and a small rectangle on it that maps to the screen, and a far plane with a larger rectangle. These three are on a pyramid. The following image helps explain this better:



gluPerspective is easier to use and should be preferred.

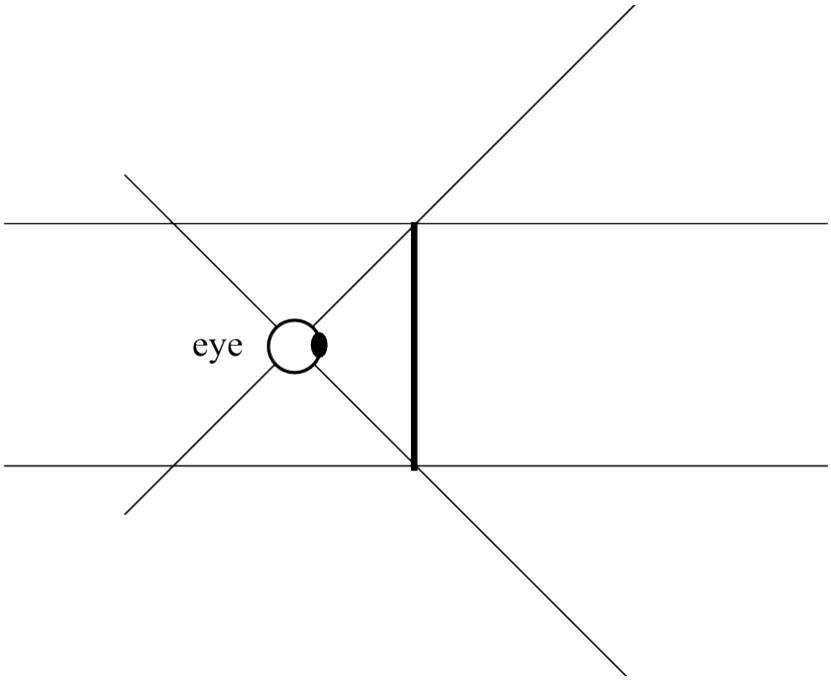
gluPerspective(fovy angle, aspect ratio, neardistance, fardistance).

Fovy angle is the angle between the top and the bottom of the pyramid. Aspect ratio is the width/height of the rectangle. Neardistance is the distance of the near plane to the eye, where we start seeing things. Far distance is the distance to the far plane that we don't see beyond. After this, we are still looking towards the –z axis.

Therefore, if we start with an identity matrix in our projection matrix, and call

gluPerspective(90, 1.0, 0.1, 100.0)

Then what we have at the z=-1 plane stays the same, while everything else scales:



However, it is better to set a fov angle that matches the approximate distance of your eye to the screen and the size of the screen. 40 can be a good approximation.

You usually set the projection matrix once and don't touch it afterwards:

```
void init() {
      glMatrixMode(GL_PERSPECTIVE);
      glLoadIdentity();
      gluPerspective(40, 1.0, 0.1, 100.0);
      glMatrixMode(GL_MODELVIEW);
}
```

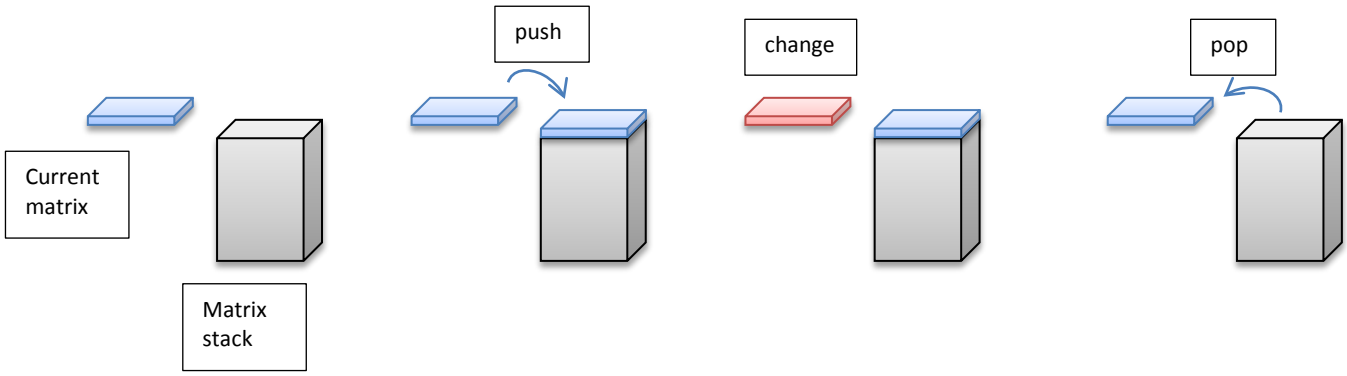
We will see how to move the camera later below.

The matrix stack

Note that in OpenGL the only matrix operations you can do with the current matrix are:

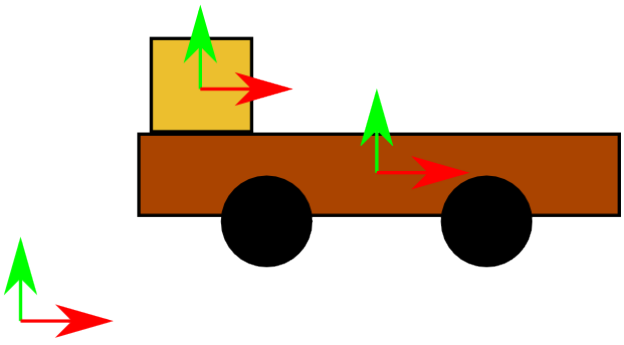
1. glGet*: read the current matrix
2. glLoad*: set the current matrix to be a specific matrix
3. glMult, glRotate, glTranslate, etc: postmultiply the current matrix with a new matrix ($M_{current}$=$M_{current}$.M_{new})

Sometimes, you want to remember the value of the current matrix before you change it to something else (we will see why we would want that soon below). While you could do glGet and save it to somewhere and then do glLoad later, glGet is a very inefficient OpenGL command as it stalls the GPU's pipeline to read the current value of the matrix. Instead, we can tell the GPU to remember the current matrix for later use. We do this by pushing the matrix to the matrix stack with glPushMatrix() and popping it out later with glPopMatrix(). This way the matrix does not have to leave the GPU, which does not cause inefficiency. Let's see how this works:

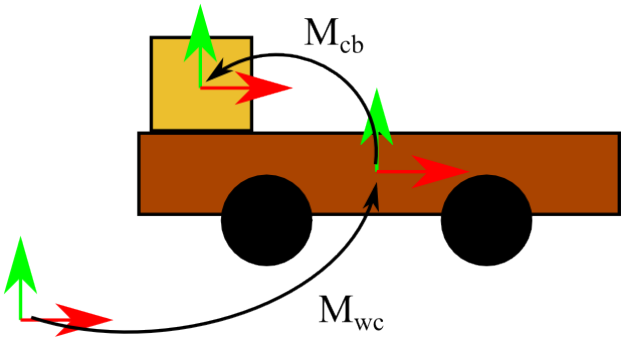


When we want to remember the value of the current matrix later, we push it to the matrix stack. After that, we can change the current matrix freely, postmultiply it, etc. Later, when we want to go back to that old value, we pop it from the top of the stack. Note that you can pop only once for every push. For every push there should be one pop, so that your program functions correctly.

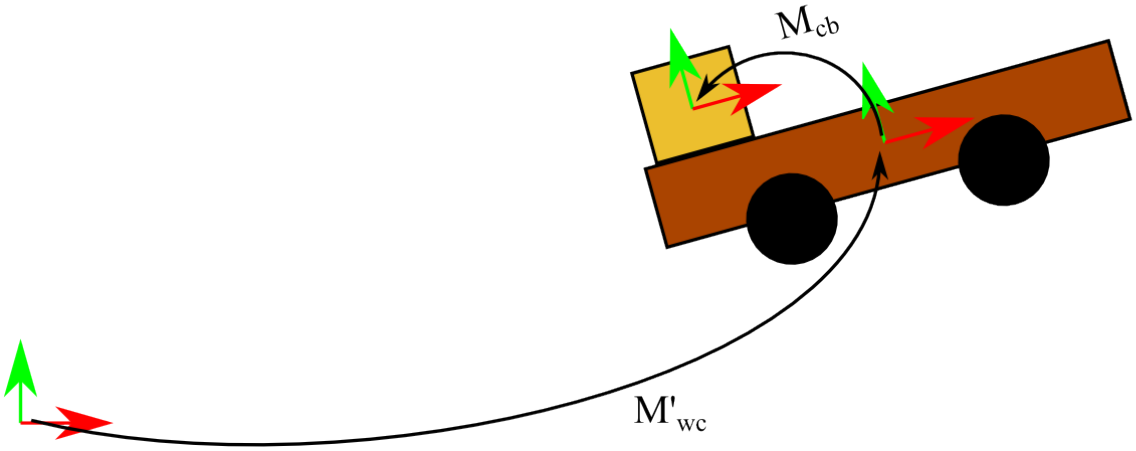
Let's see why we might want to remember the value of a matrix. Similar to the house and the tree example, let's assume that our artist created a cart and a box, and we placed them in our world like this:



We color the x axes with red and y axes with green. We see the world coordinate frame, as well as the local frames of the two objects. When the cart moves, we want the box to move with it. Therefore, we define the box's transformation relative to the cart:



The matrix M_{wc} is the translation matrix that brings the cart from the world's origin to where it is now. The matrix M_{cb} is the translation matrix that brings the box from the cart's origin to where it is now. Therefore, the box's transformation relative to the world is the matrix multiplication $M_{wc}.M_{cb}$ This way, if we change M_{wc}, we can keep M_{cb} constant and M_{cb} would move with the cart:



The new matrix for the cart is M'_{wc}. Since we calculate the world matrix of the box as $M'_{wc}.M_{cb}$, we didn't have to change the matrix of the box relative to the cart. If we were to write OpenGL code to draw this it would look something like this:

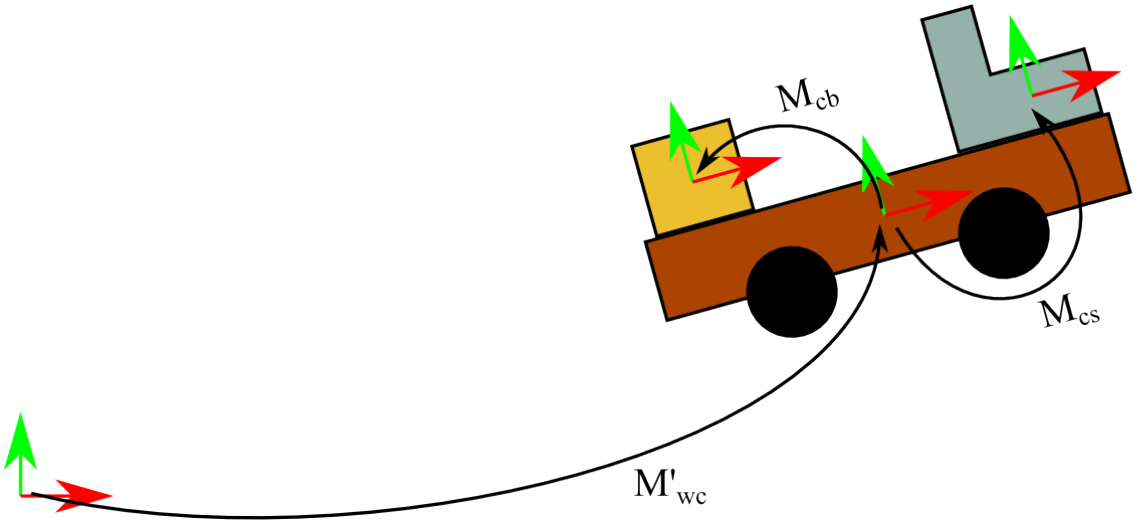
```
glMultMatrix(Mwc)
drawTheCart()
glMultMatrix(Mcb)
drawTheBox()
```

Note that instead of glMultMatrix() we usually use a series of glTranslate() and glRotate() calls in actual programs.

As you can see, we are multiplying the current matrix as we go along. If we assume that initially the current matrix was identity, here is the current value of the matrix as we go along:

```
glMultMatrix(Mwc)     //current value of the matrix became Mwc
drawTheCart()         //the cart is drawn as Mwc.p
glMultMatrix(Mcb)     //current value of the matrix became Mwc.Mcb
drawTheBox()          //the box is drawn as Mwc.Mcb.p
```

This is great. What if we also had a seat on the cart?



Everything else is the same, and the seat's matrix is $M'_{wc}.M_{cs}$, that is, the cart's matrix relative to the world, multiplied by the seat's matrix relative to the cart. Let's see how we could implement that in OpenGL:

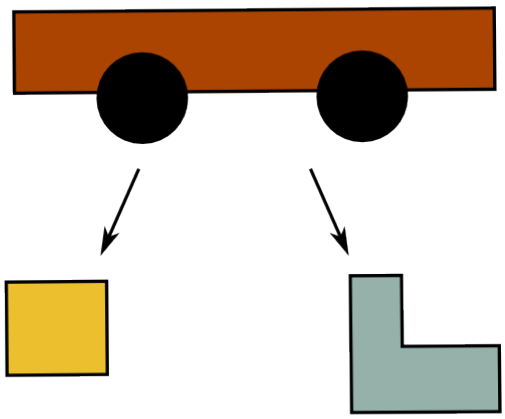
```
glMultMatrix(Mwc)     //current value of the matrix became Mwc
drawTheCart()         //the cart is drawn as Mwc.p
glMultMatrix(Mcb)     //current value of the matrix became Mwc.Mcb
drawTheBox()          //the box is drawn as Mwc.Mcb.p
glMultMatrix(Mcs)     //current value of the matrix became Mwc.Mcb.Mcs
drawTheSeat()         //the seat is drawn as Mwc.Mcb.Mcs.p
                      //but it should be Mwc.Mcs.p :(
```

As you can see, we have a problem. Since the current matrix is $M'_{wc}.M_{cb}$, if we multiply it by M_{cs}, we get $M'_{wc}.M_{cb}.M_{cs}$. However, as we said above, we need it to be $M'_{wc}.M_{cs}$ for the seat. If only we could "undo" the **glMultMatrix(Mcb)** step, everything would be fine. Then, the current matrix would go back to being M'_{wc}, and **glMultMatrix(Mcs)** would make it $M'_{wc}.M_{cs}$ like we needed. This "undo" mechanism that we need is exactly why we need the matrix stack. Here is how we would fix this code:

```
glMultMatrix(Mwc)      //current value of the matrix became Mwc
drawTheCart()          //the cart is drawn as Mwc.p
glPushMatrix()         //I know I'll need this matrix later
glMultMatrix(Mcb)      //current value of the matrix became Mwc.Mcb
drawTheBox()           //the box is drawn as Mwc.Mcb.p
glPopMatrix()          //current value of the matrix became Mwc again :)
glMultMatrix(Mcs)      //current value of the matrix became Mwc.Mcs :)
drawTheSeat()          //the seat is drawn as Mwc.Mcs.p :)
```

As you see, we used glPushMatrix() to store the current matrix on top of the matrix stack, and later retrieved it back using glPopMatrix().

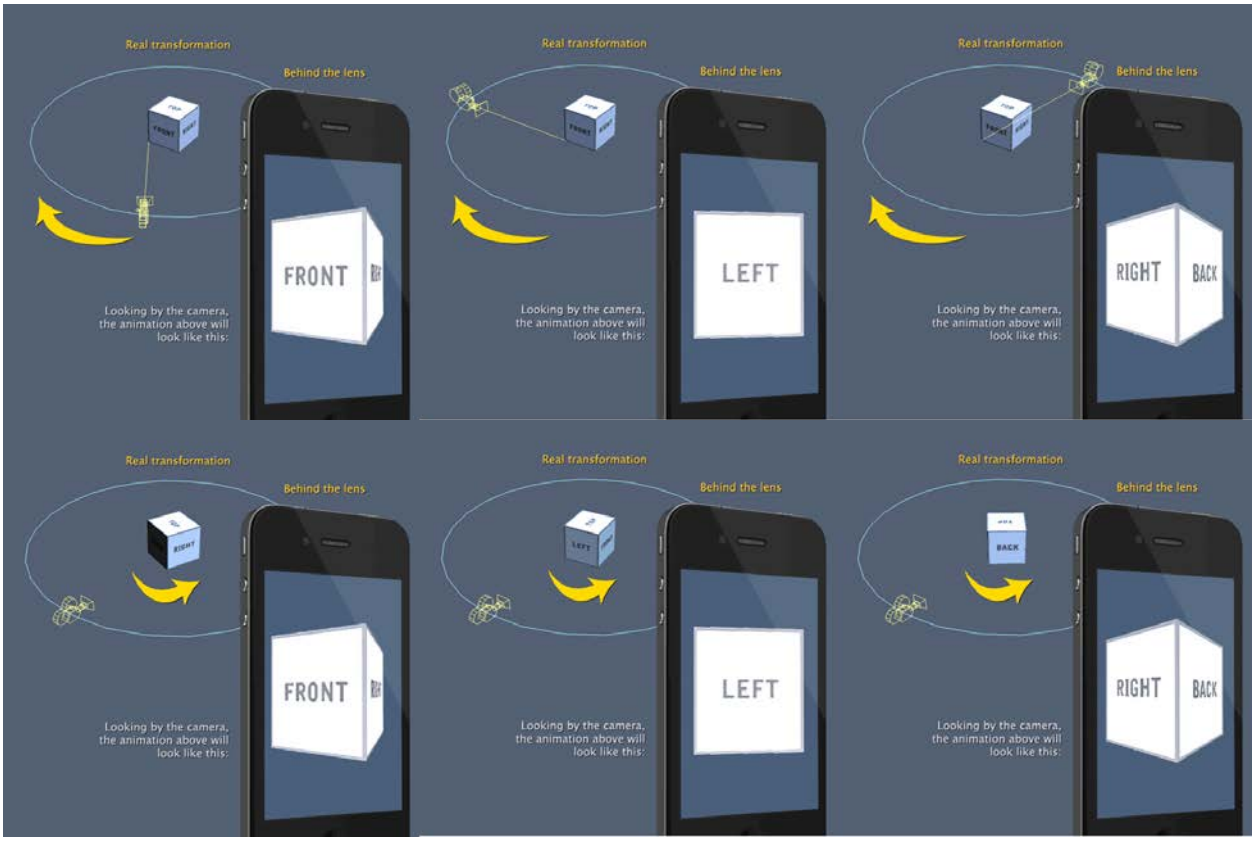
If we draw the relationship between our cart, box and the seat conceptually,



This tree (or more accurately Directed Acyclic Graph) structure is called a scene graph. Scene graphs indicate the parent/child relationships of objects in the scene. They are rendered using a depth-first traversal, with a glPushMatrix() every time we go down a level and a glPopMatrix() every time we go up. This way we can have scene graphs with arbitrary structures. When you are drawing more than one object in a scene, try to think of it like a scene graph, even if you do not represent your data like a graph.

Camera Manipulation

Once you write the code to render your scene, you may want to change the way you look at the scene. Maybe you want to walk around the cart from the last example and look at it from different directions.



[Images are borrowed from http://db-in.com]

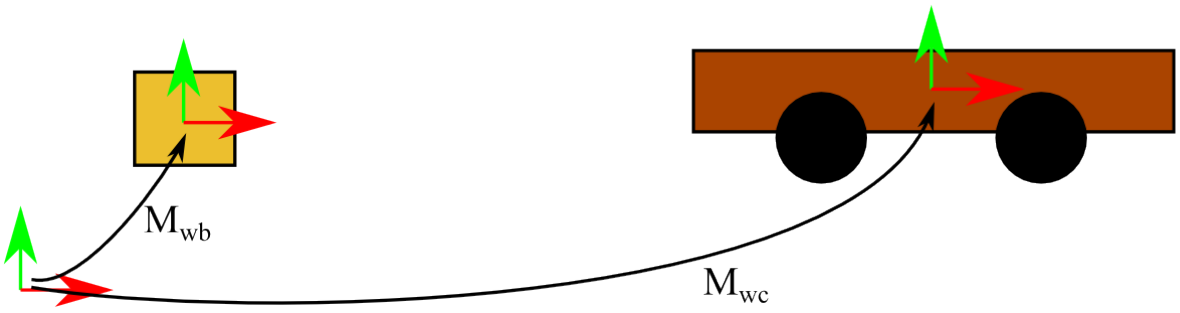
On the top, we see a camera rotating around the cube, and we see what is being rendered on the screen. On the bottom, we see the camera staying still and everything else in the world (in this case, only the cube) rotating. As you can see, the two operations result the same rendered images. Therefore, this is how we will implement camera motions, we will fake it by moving everything else.

Recall that we said we can use the projection matrix for camera position. While this is a correct way of implementing camera motions, the convention is to use the projection matrix only for the actual projection transformation (perspective, orthogonal) and implement the camera motions in the model*view* matrix.

To summarize, we have a projection matrix M_p that defines our camera volume centered at the origin and a *model* matrix M_m for our object (like $M'_{wc}.M_{cs}$ for the seat on the cart example). The matrices are multiplied as $M_p.M_m$ before the seat is rendered on the screen. If we add a camera, or *viewing* matrix M_v in between, it would become $M_p.M_v.M_m$. As you can see, since M_v is to the left of M_m, it is more on the outside. By locating M_v right after M_p, we ensure that everything in the world is transformed with M_v. Note that we can either take $M_p.M_v$ as our GL_PROJECTION matrix and not worry about the camera in

our modelview matrix, or take $M_v.M_m$ as our modelview matrix and always ensure that the modelview matrix contains the camera matrix first before we start multiplying it in our OpenGL code. The first method is easier, but not the convention (it is not called projectionview matrix).Therefore, we will use the second method.

Let's see this with an example. Let's say we have our cart and the box is on the ground to keep things simpler.



If our camera (or viewing) matrix is M_v, we would render these objects with the modelview matrices $M_v.M_{wb}$ and $M_v.M_{wc}$. We could implement this as either setting the camera matrix before each:

```
locateCamera()        //sets the current value of the matrix to Mv
glMultMatrix(Mwc)     //current value of the matrix became Mv.Mwc
drawTheCart()         //the cart is drawn as Mv.Mwc.p
locateCamera()        //sets the current value of the matrix to Mv
glMultMatrix(Mwb)     //current value of the matrix became Mv.Mwb
drawTheBox()          //the box is drawn as Mv.Mwb.p
```

Or we can push the camera matrix to the matrix stack:

```
locateCamera()        //sets the current value of the matrix to Mv
pushMatrix()          //push Mv to the matrix stack
glMultMatrix(Mwc)     //current value of the matrix became Mv.Mwc
drawTheCart()         //the cart is drawn as Mv.Mwc.p
popMatrix()           //current value of the matrix became Mv
```

```
pushMatrix()          //we have to push it again or we can't pop
glMultMatrix(Mwb)     //current value of the matrix became Mv.Mwb
drawTheBox()          //the box is drawn as Mv.Mwb.p
popMatrix()           //current value of the matrix became Mv
                      //for drawing future objects
```

The locateCamera() function would load the identity matrix to the modelview matrix, and do the necessary translations and rotations that would move everything in the world to give the impression that the camera has moved.

Combining Transformations

Note that matrix multiplication is not commutative. That is, M1.M2 is usually not equal to M2.M1. This is true if any of the matrices do any rotation. Let's say Mr is a rotation matrix and Mt is a translation matrix. Mr.Mt and Mt.Mr transform the cart differently. Let's first investigate what it means to combine transformations this way.

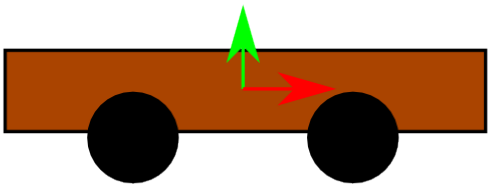
If we transform our cart with Mr.Mt, we can think of it two ways:

(1) we first rotate the cart (Mr), and then translate it in the rotated coordinate frame (Mr.Mt) (going inwards),

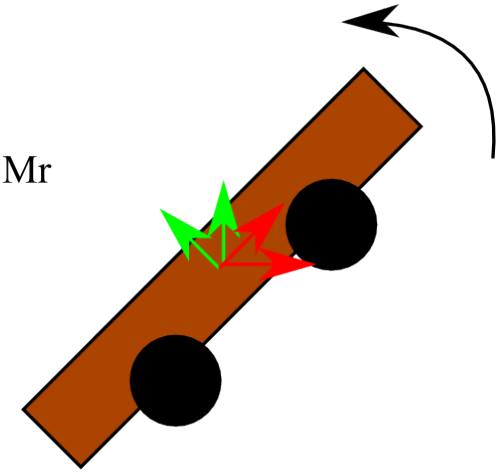
(2) we first translate the cart (Mt), and rotate the whole world around it (Mr.Mt) (going outwards).

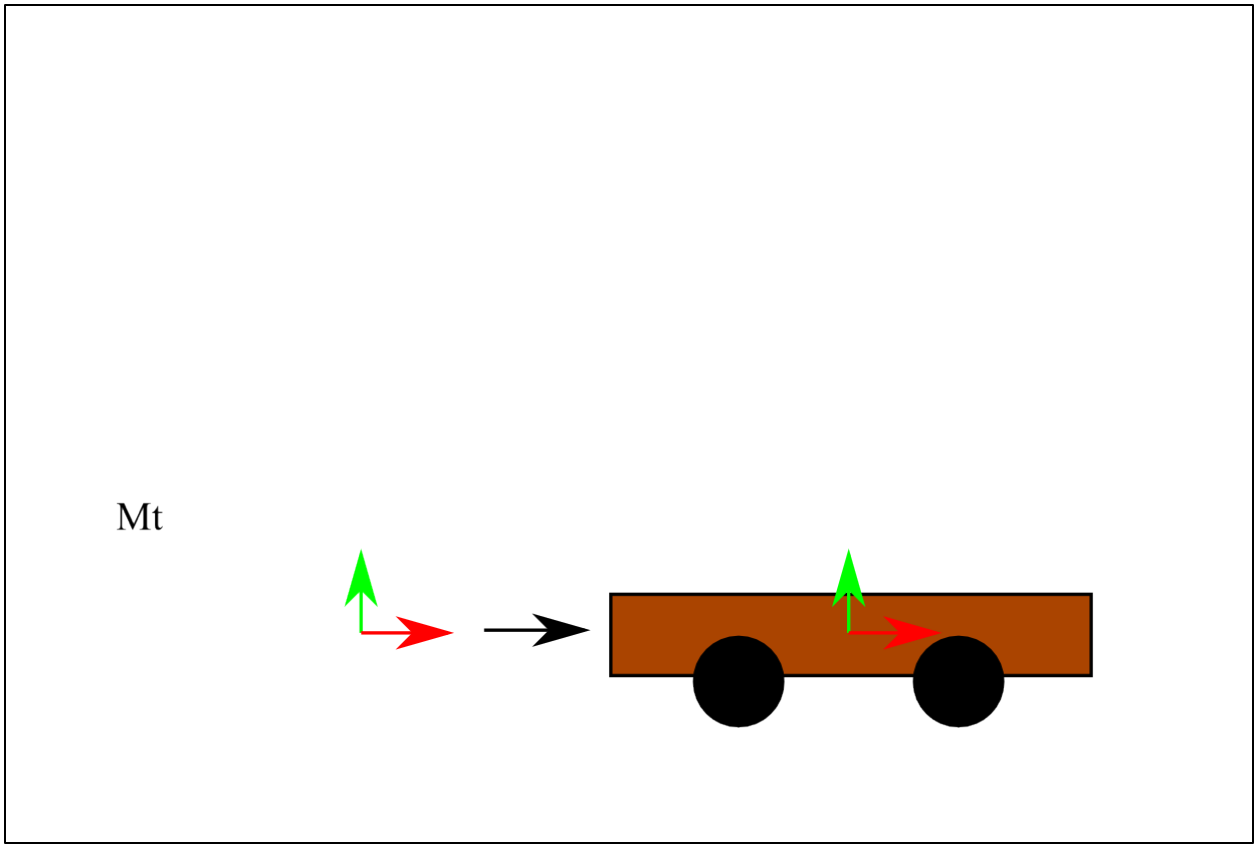
While you are writing OpenGL code, you usually think of it as the first one. However, when dealing with the camera, you may want to think of it as the second way because the viewing matrix always goes first. Let's assume that Mr rotates 45 degrees counter-clockwise, and Mt moves along the positive X axis and let's see (1) and (2) in examples. First let's see what these transformations do by themselves:

Identity matrix, at world's origin.



Mr



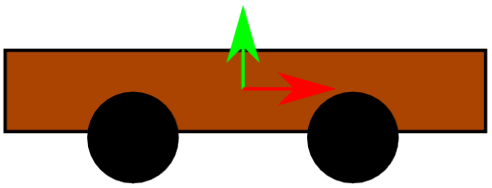


As we can see, Mr turns 45 degrees counterclockwise (around the Z axis) and Mt translates along the positive X axis with a distance around the length of the cart.

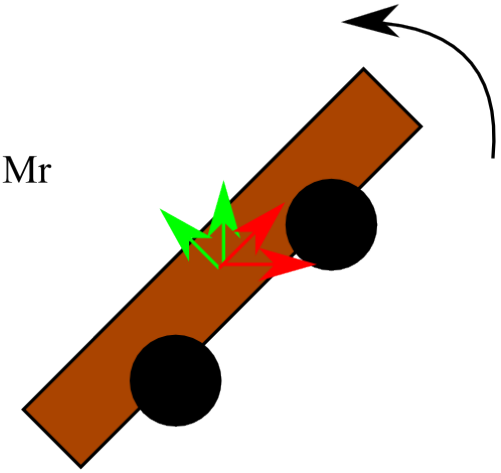
Then we see (1), which was :

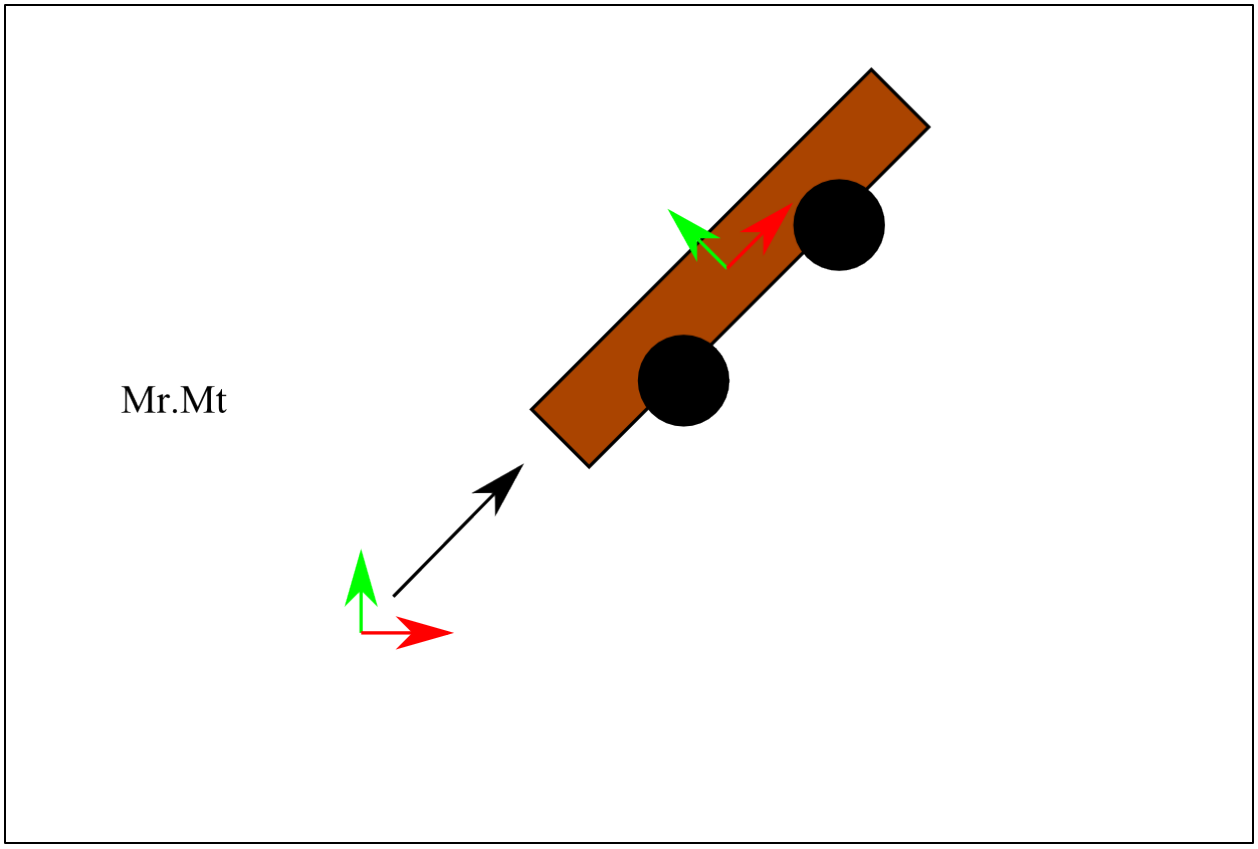
(1) we first rotate the cart (Mr), and then translate it in the rotated coordinate frame (Mr.Mt) (going inwards),

Identity matrix, at world's origin.



Mr



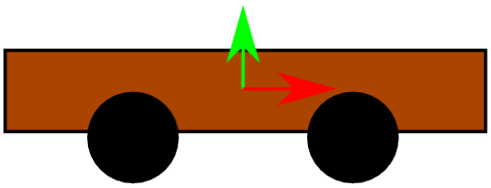


As you can see, since Mt was multiplied on the right, it is an “inner” transformation. Therefore the cart is transformed in the local coordinates. Since Mt is supposed to move the cart along the X axis, it moves it in the local X axis, which is 45 degrees turned with the cart.

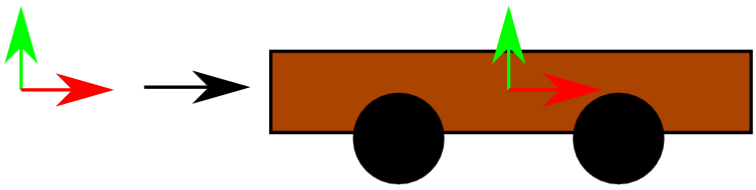
Now let’s see the second case (2), which was:

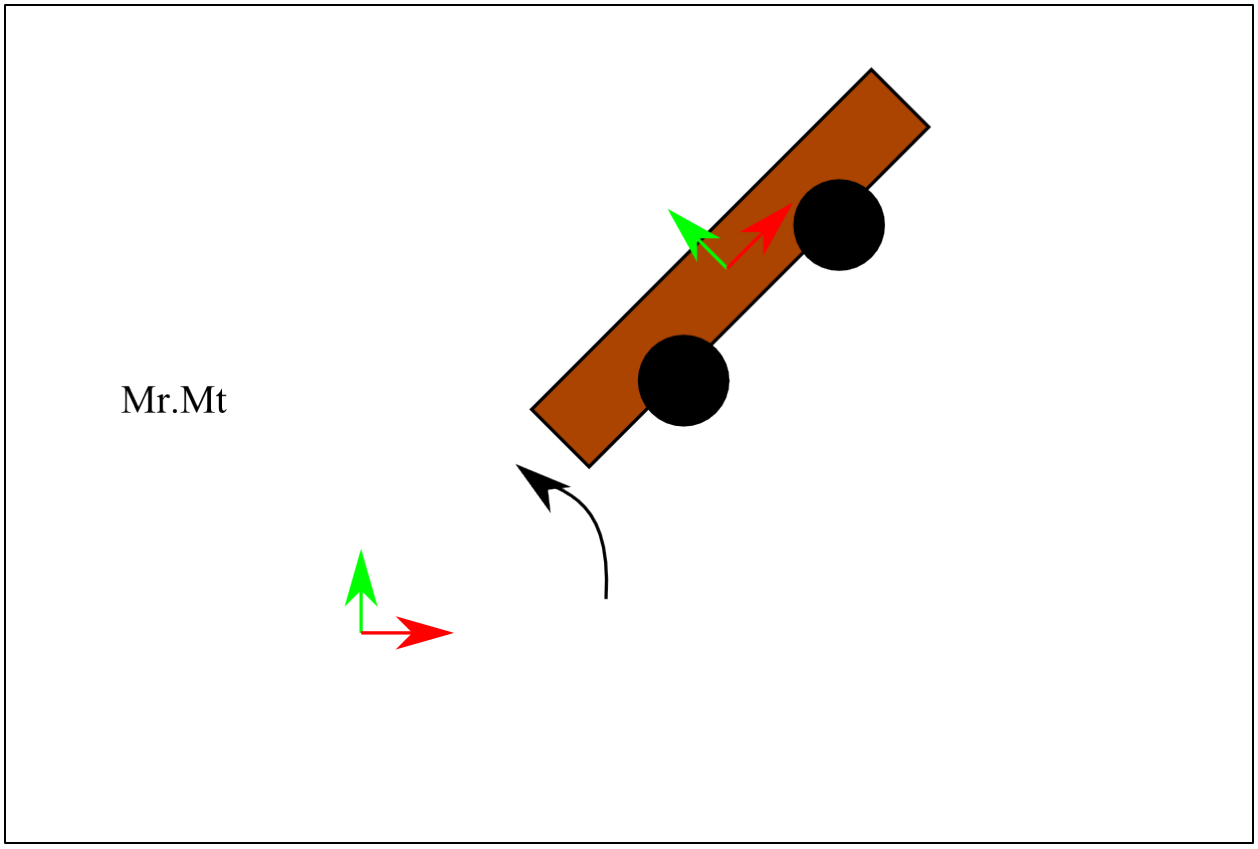
(2) we first translate the cart (Mt), and rotate the whole world around it (Mr.Mt) (going outwards).

Identity matrix, at world's origin.



Mt

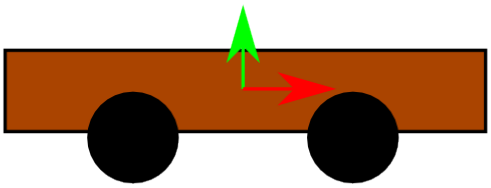




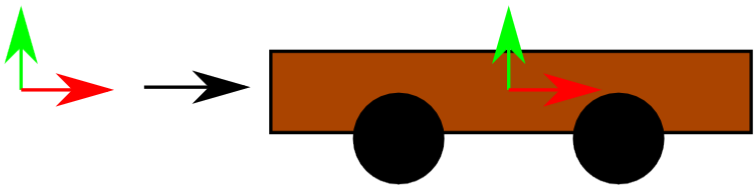
As you can see, since Mr was multiplied on the left, it is an “outer” transformation. Therefore the cart is transformed in the word coordinates, around the world’s origin. Since Mr is supposed to rotate the cart 45 degrees counterclockwise, it rotated the whole world, including the translation of the cart.

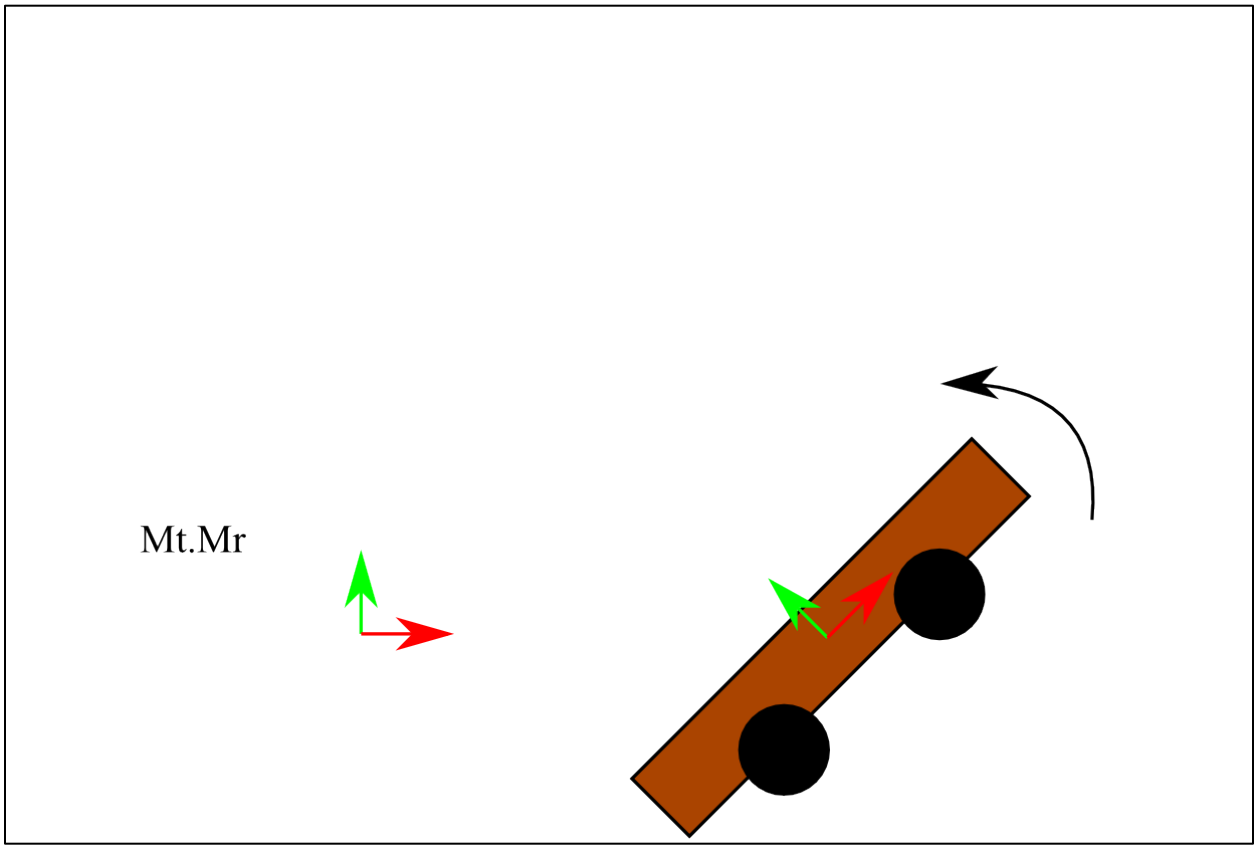
This was Mr.Mt. Let’s see what Mt.Mr does. First going from world to local like (1):

Identity matrix, at world's origin.



Mt

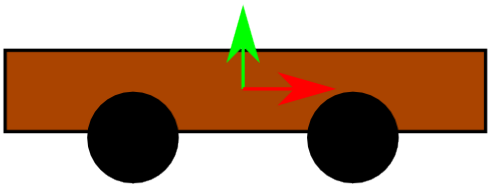




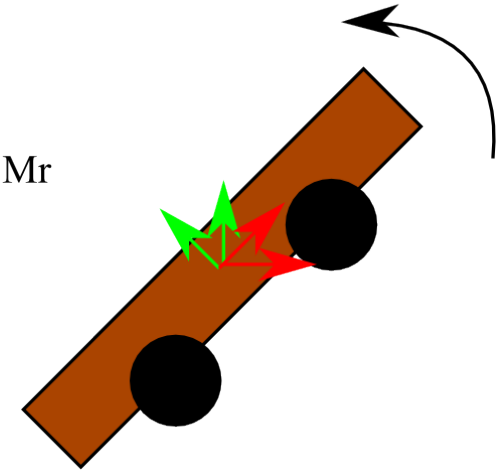
As you can see, since Mr was multiplied on the right, it is an “inner” transformation. Therefore the cart is transformed in the local coordinates. Since Mr is supposed to rotate the cart 45 degrees counterclockwise, it rotates the cart in the local coordinate frame, around the local origin.

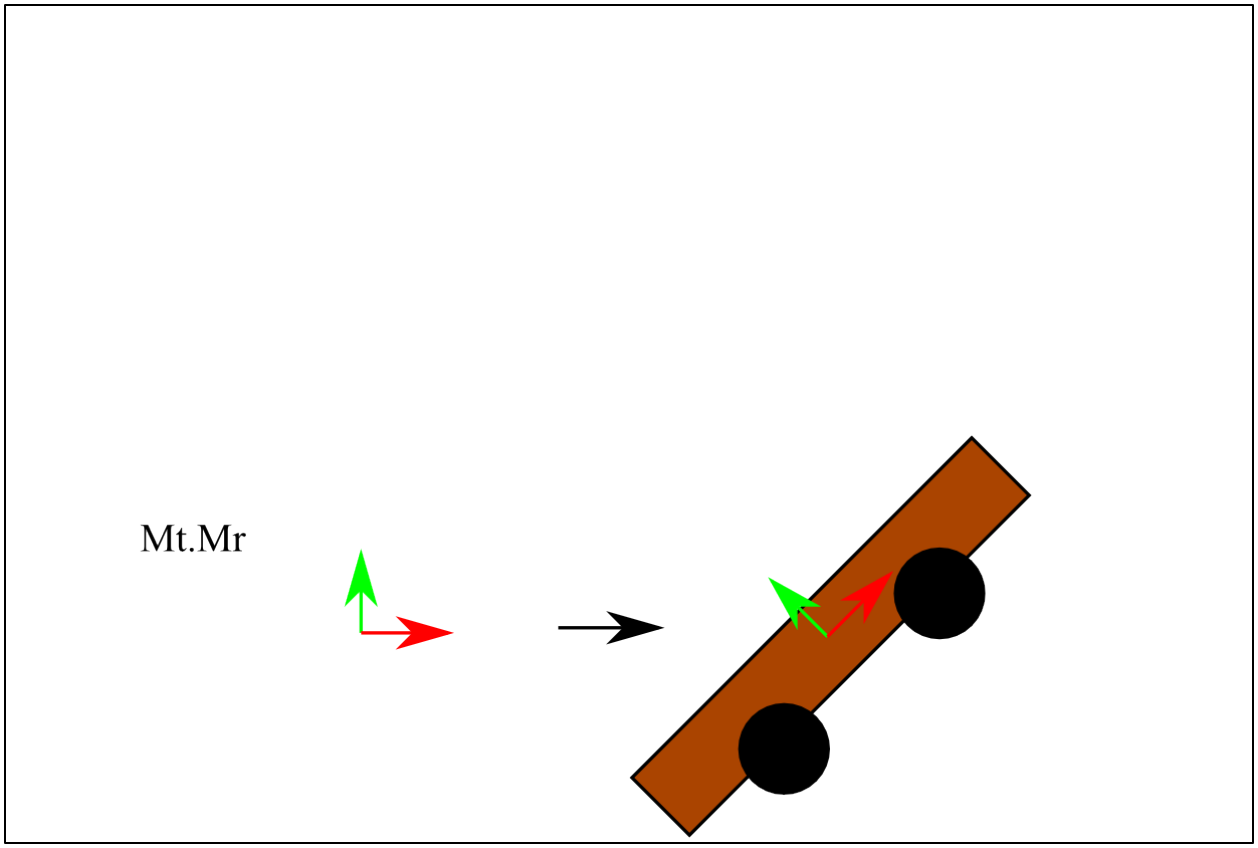
Now let’s go from local to world like (2):

Identity matrix, at world's origin.



Mr





As you can see, since Mt was multiplied on the left, it is an “outer” transformation. Therefore the cart is transformed in the word coordinates, along the world’s x axis. Since Mt is supposed to move the cart along the X axis, it moves it in the world X axis, which is not turned.

Note how Mr.Mt and Mt.Mr are different.

Hopefully, this makes clear the ordering of matrices and what we mean by “local coordinates” and “world coordinates”.

Now that we know how to combine transformations, we can think about how we can set up the camera matrix such that the camera rotates around the object like the earlier figures with the iPhone picture.

Note that in that transformation, the world is first moved away from the camera’s origin, and then rotated in a local coordinate frame. Therefore our camera matrix becomes Mv=Mt.Mr.

Our program can keep track of two variables: cameraDistance and cameraAngle. Then in the locateCamera() function, it can use them as:

```
void locateCamera() {
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
```

```
        glTranslatef(0.f, 0.f, -cameraDistance);
        glRotatef(cameraAngle, 0.f, 0.f, 0.f); //correct this
                                    //as an exercise please :)
}
```

This way we load the camera matrix before multiplying it with the model matrices.

To be continued…

Continued on Week 7:

Pre-lecture:

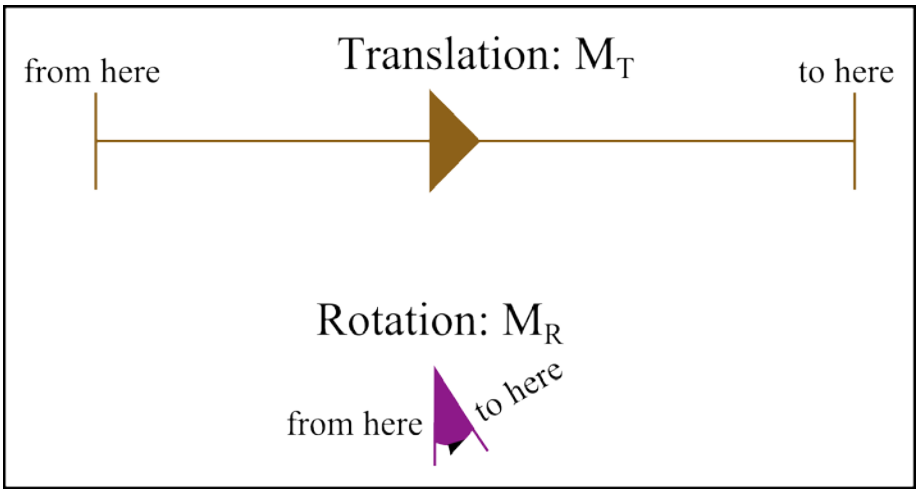
- Ask to class to define: Transformation, Translation, Rotation, point, transformation matrix
- Important that you learn in the lectures, not in the lab
- Make sure you are in the Google group

Intuitive Understanding of Transformations

Previously we have seen how transformations work and how they are combined. Here we will see more concrete and easy-to-visualize examples. This will help us have a more intuitive understanding of transformations, how they work and what it means to combine them in different ways. Since transformations were abstract entities, it may have been difficult to understand. Now we will use concrete physical entities that represent transformations and will use them to understand transformations better.

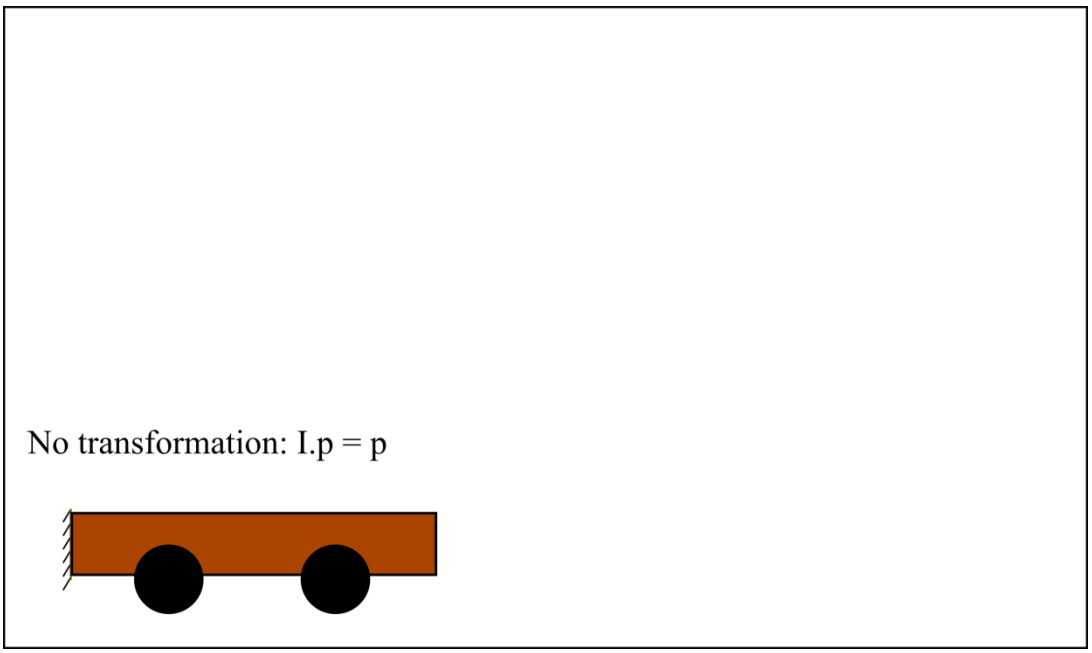
Let us imagine transformations to be physical connectors between objects in the space. You can imagine translations to be straight connectors with both ends having the same orientation and rotations to be

small connectors that simply make the next object have a different orientation.

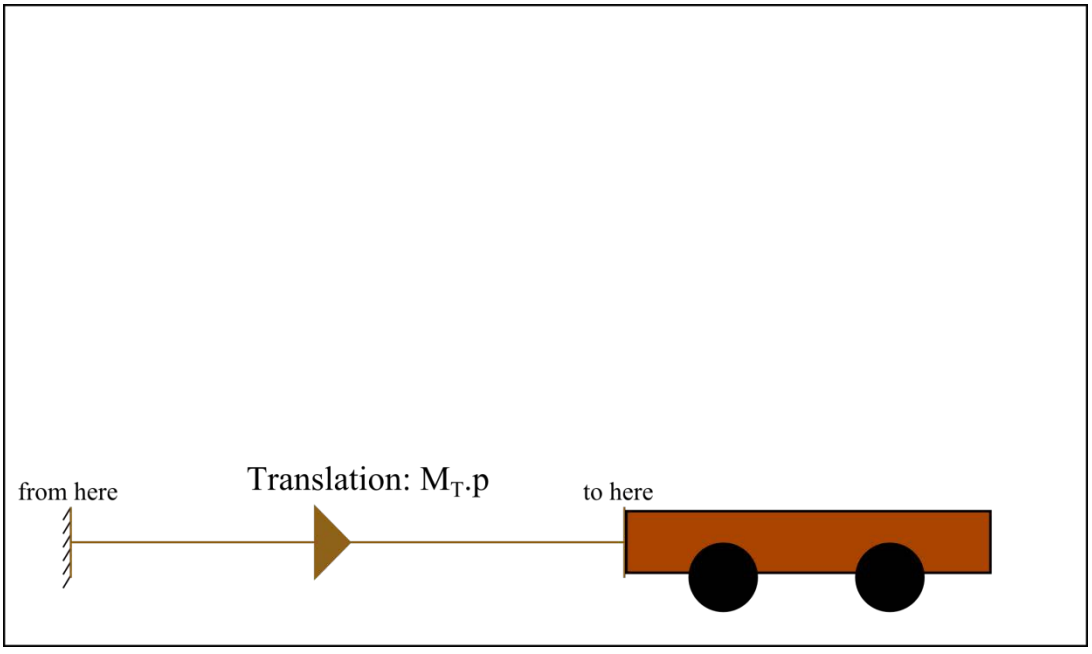


Imagine the brown lines to be a physical separator between two objects. An object can be attached each to the “from here” part or the “to here” part. Similarly, the pink object is also a separator but the objects on the two ends would have different orientations. Let’s see how this works with our cart example:

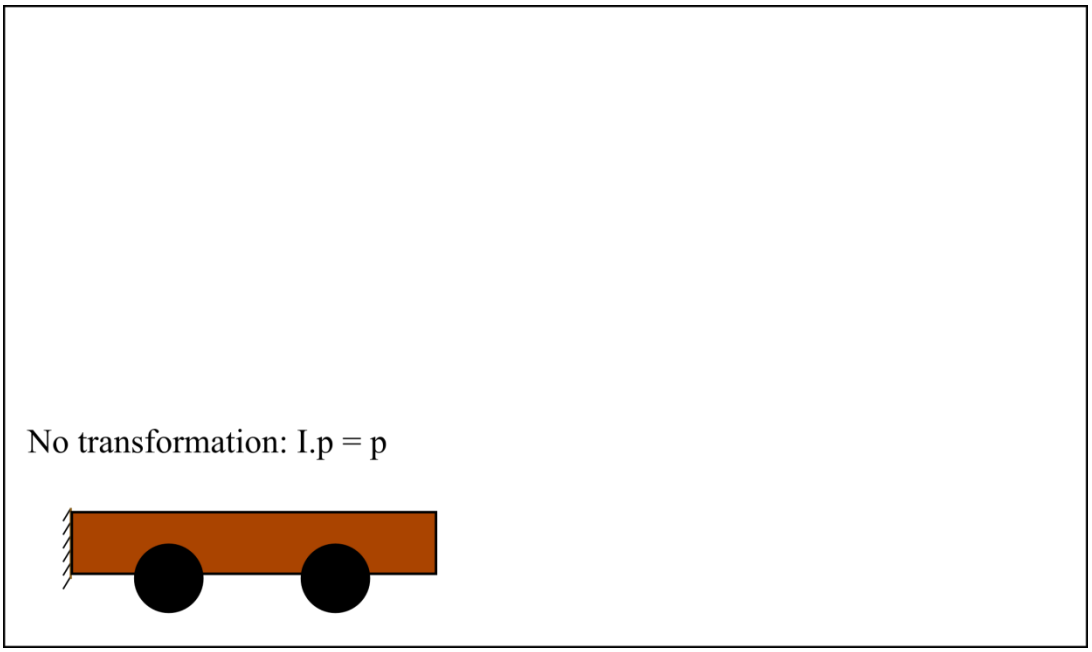
In our examples, we will attach things to a wall on the bottom left of the image. First with no transformation, our cart is attached to the wall:



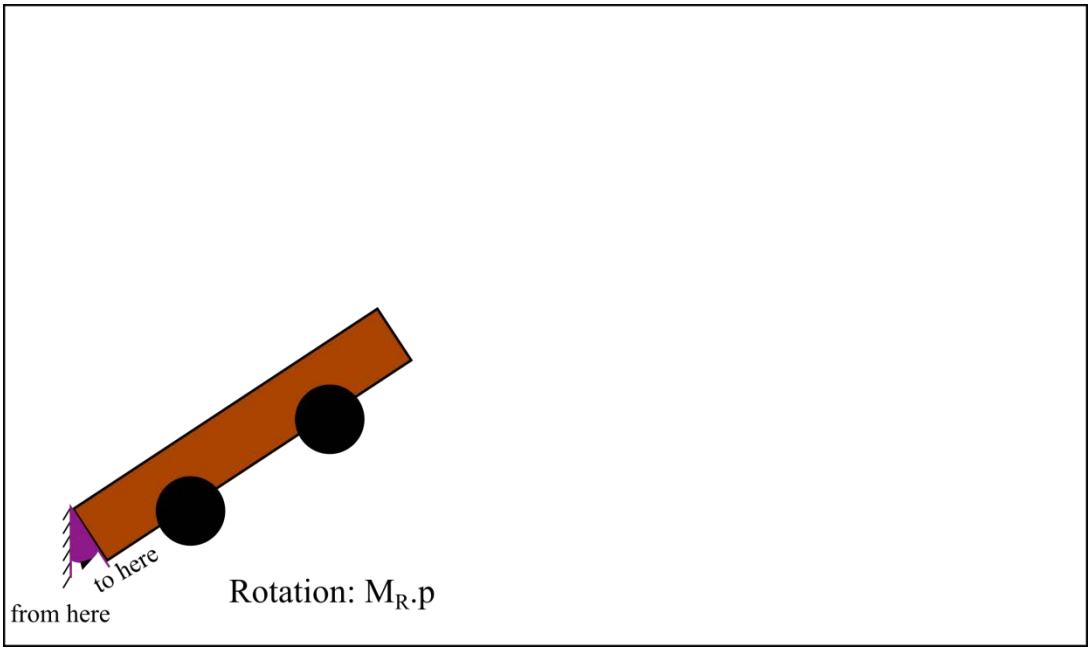
If we apply the translation to that, we attach the translation connector on the wall and attach the cart to the connector. This takes the cart to the right:



Now we will use the rotation connector. Let's remember the no transformation case again:

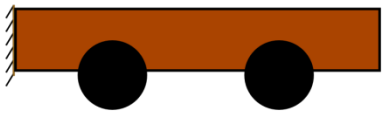


If we use the rotation connector, we attach the rotation connector on the wall, and attach the cart to the rotation connector. As a result, the cart rotates upwards:



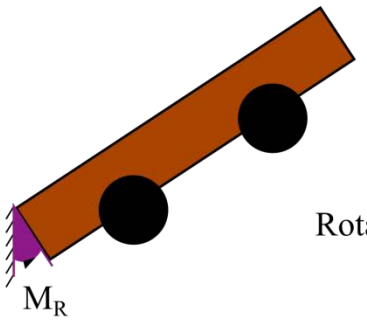
This is basically what transformations in computer graphics do. We represented them with solid objects to make it more intuitive. This representation especially makes combining transformations more intuitive. Let's see a basic example:

We start with the empty wall, which is no transformation:



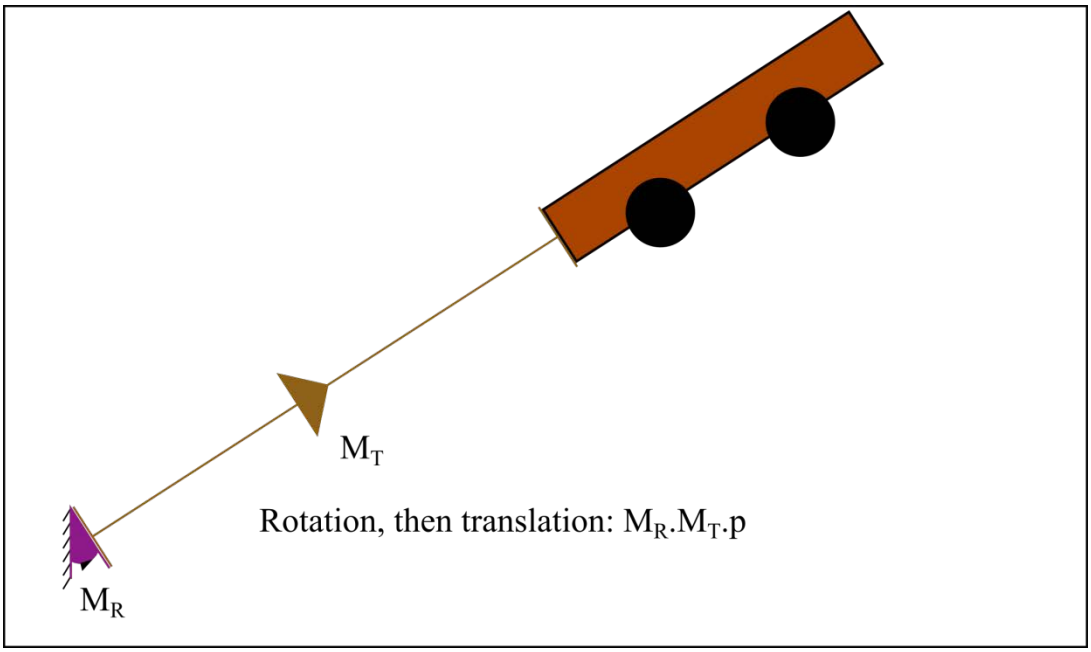
No transformation: I.p = p

Then we add a rotation to it:

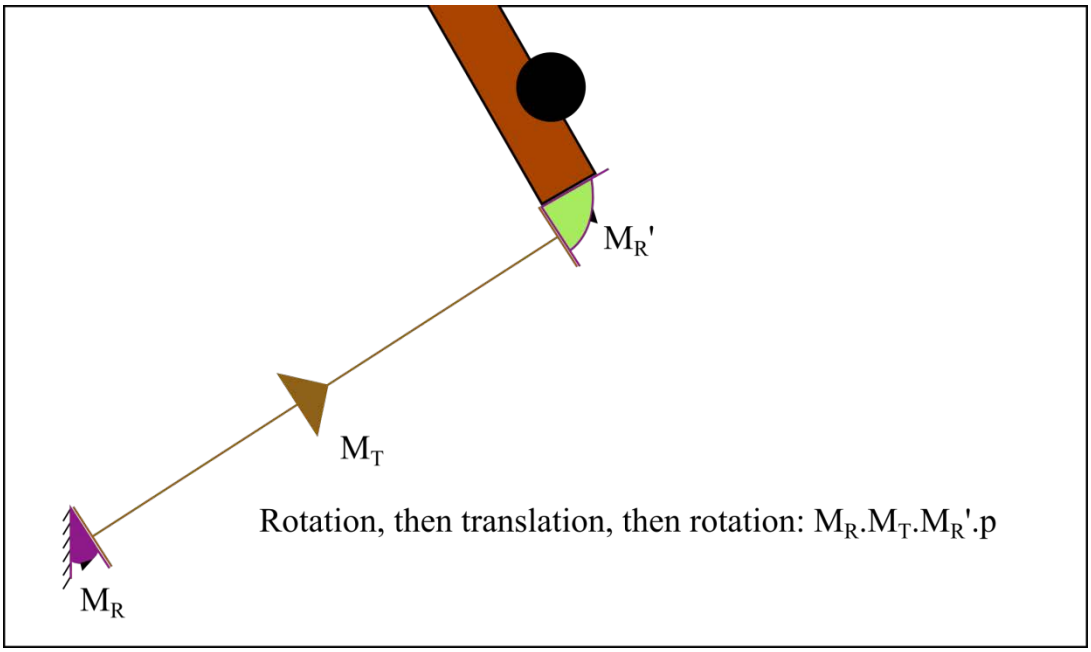


Rotation: M_R.p

Then we add a translation to it:



Then let's add another rotation to it:



As you can see, we pushed the cart further and further from the wall. Each time we added a new transformation between the cart and the previous transformation that we added in the previous step. Here is more or less what happened, with the newly added transformation in each step in **bold**:

Wall -> Cart

Wall -> **M_R** -> Cart

Wall -> M_R -> $\mathbf{M_T}$ -> Cart

Wall -> M_R -> M_T -> $\mathbf{M_R'}$ -> Cart

As you can see, we always added the transformation right before the cart and after the previously added one. If you follow the transformation matrices in the pictures, here is how it evolved:

p

$\mathbf{M_R}$.p

M_R.$\mathbf{M_T}$.p

M_R.M_T.$\mathbf{M_R'}$.p

As you can see, similarly the newly added matrix was always before p (a point of the cart) and the previously added transformation.

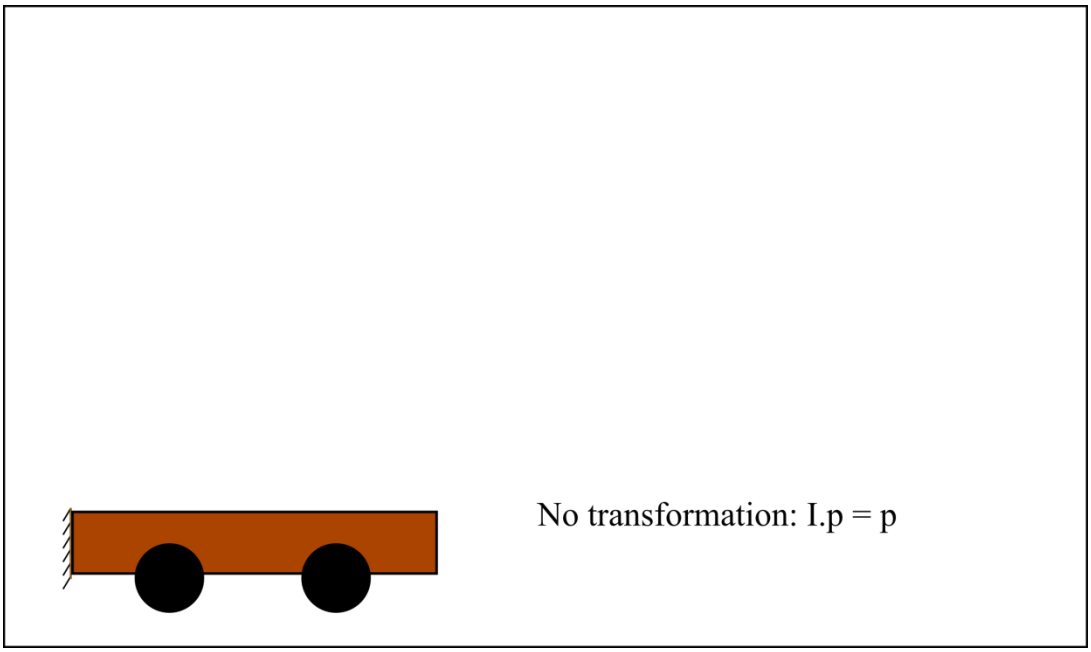
This is what we meant when we said “inner” transformations. As we keep adding transformations like that, those transformations are applied to the end of the transformed chain, which is an “inner” attachment place, not the “world” wall that we had down below on the left.

You can think of this in terms of world coordinates and local coordinates. The wall on bottom left is the world, or the origin of the world coordinates. As we attach transformations, what we attach after those transformations are attached to a “local” wall, that is the end of the transformation piece. That is local coordinates.

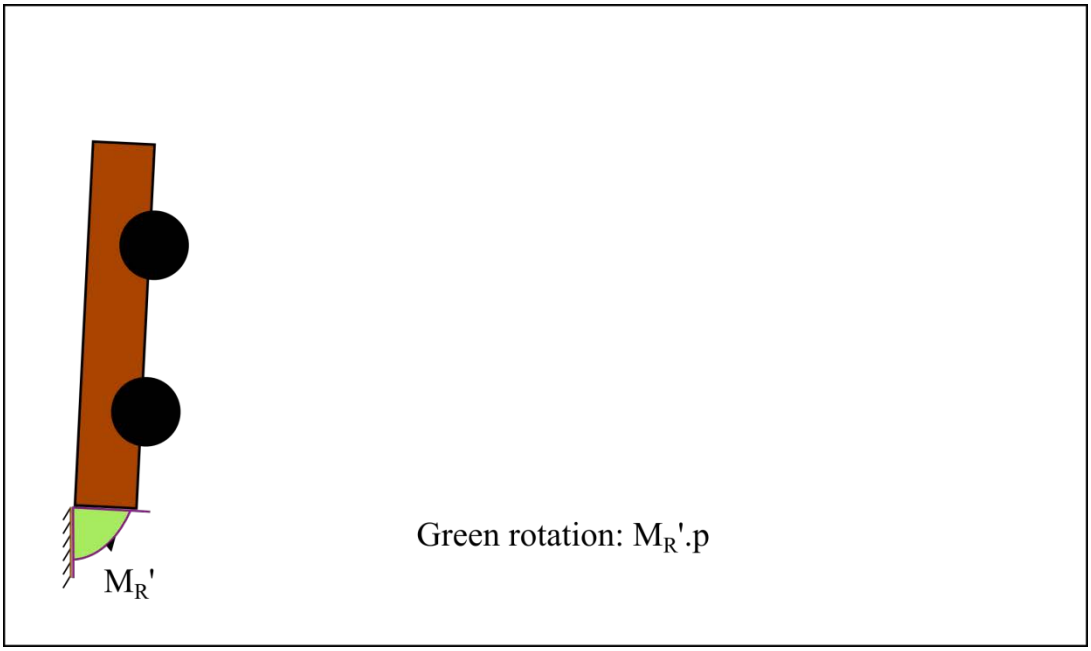
By adding transformations at the end and right before the card, we applied transformations in the “cart’s local coordinates” or “local transformations”. Because that was where the cart was, and instead we attached a transformation there.

We could also attach transformations in the “world coordinates” or apply “world transformations”. Since the world is the wall, this would mean that we attach newly added transformations between the wall and whatever is attached to the wall. Let’s see the same example by adding world transformations each time.

Again, we start with our cart on the wall:

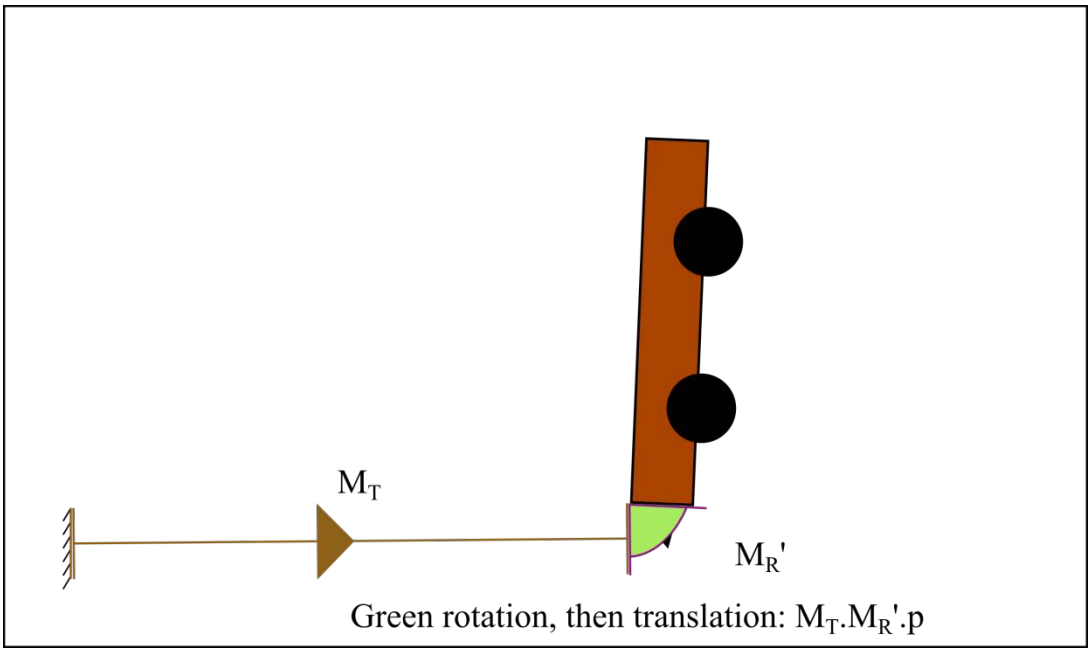


Then, let's add the green rotation between the cart and the wall:



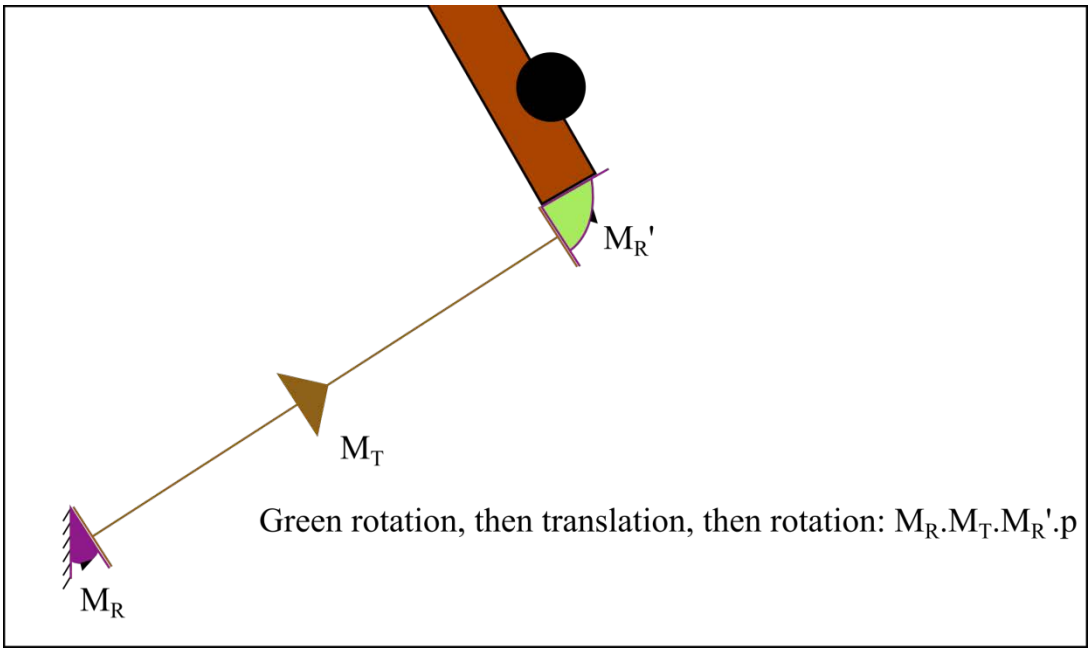
Note that even though the transformation that we used was different, this step was exactly the same as our previous example. This is because when there was no transformation, the cart's local coordinates were equal to the world coordinates. This made world and local transformations be the same thing. If you do not understand this now, **come back here later after you saw everything**.

Anyway, let's add a translation in the world coordinates:



Since we wanted to transform the cart in world coordinates, we inserted our translation next to the wall (between wall and green rotation). If we wanted to transform the cart in local coordinates, we would insert our translation next to the cart (between cart and green rotation). Understand that the two move the cart differently (local would move the cart upwards, **make sure you understand why**).

OK, let's apply one more transformation in world coordinates. This time, let's add our purple rotation in world coordinates:



Similarly, since we wanted to apply the transformation in world coordinates, we inserted it next to the wall. Note that our cart ended up in the exact same place as it was in page 38, even though we added the same transformations but in the inverse order.

Here is more or less what happened, with the newly added transformation in each step in **bold**:

Wall -> Cart

Wall -> $\mathbf{M_R'}$ -> Cart

Wall -> $\mathbf{M_T}$ -> M_R' -> Cart

Wall -> $\mathbf{M_R}$ -> M_T -> M_R' -> Cart

As you can see, we always added the transformation right after the wall and before the previously added one. If you follow the transformation matrices in the pictures, here is how it evolved:

p

$\mathbf{M_R'}$.p

$\mathbf{M_T}$.M_R'.p

$\mathbf{M_R}$.M_T.M_R'.p

As you can see, similarly the newly added matrix was always before everything else. This is what we meant when we said “outer” transformations. As we keep adding transformations like that, the newly added transformations transformed everything that came after them, because we inserted them before the wall, where everything was attached to.

This is what we meant by “outer” transformations. By transforming everything that exists in the scene, it basically transformed that subspace from “outside”.

You can think of this in terms of world coordinates and local coordinates. The wall on bottom left is the world, or the origin of the world coordinates. The transformations that we attached to the world are world transformations that transform everything else, because where we add the transformation is where the world is.

Note however that even though the matrix multiplications were the same in the two examples, when we talked about them, we listed the transformations in the reverse order:

1. “Rotation, then translation, then green rotation”
2. ”Green rotation, then translation, then rotation”

Because in the first one we assumed that we were talking about the “outside to inside” or “world-to-local” order. In the second one, we assumed that we were talking about the “inside to outside” or “local-to-world” order.

Exercise: Using the same M_R, M_T and M_R', similarly draw these operations in the asked order:

- In local-to-world order:
  - M_R, then M_T, then M_R'
  - M_T, M_R', then M_R
- In outside to inside order:
  - M_R', M_T, M_R
  - M_T, M_R', then M_R

Make sure you understand both directions of thinking about combining transformations. Make sure you understand that the mathematics is the same in each direction, but we are simply trying to make sense of the order of operations in two different ways.

OpenGL and Combining Transformations

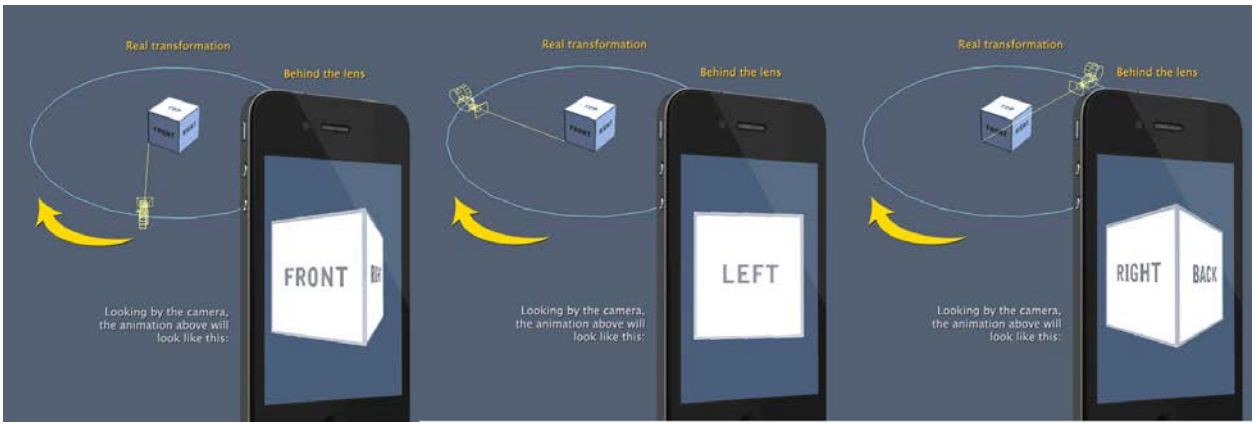
In OpenGL, you give successive commands that postmultiply the current matrix. That is, if the current matrix is M, and we call glRotate(), the current value of the matrix becomes $M.M_R$. As you recall, this is the outer-to-inner or world-to-local ordering. As you keep postmultiplying, you go more and more inside your model. Therefore, as you are writing OpenGL code, you always think of the ordering as world-to-local. The earlier matrix operations transform everything else that come after in code.

Camera Manipulation

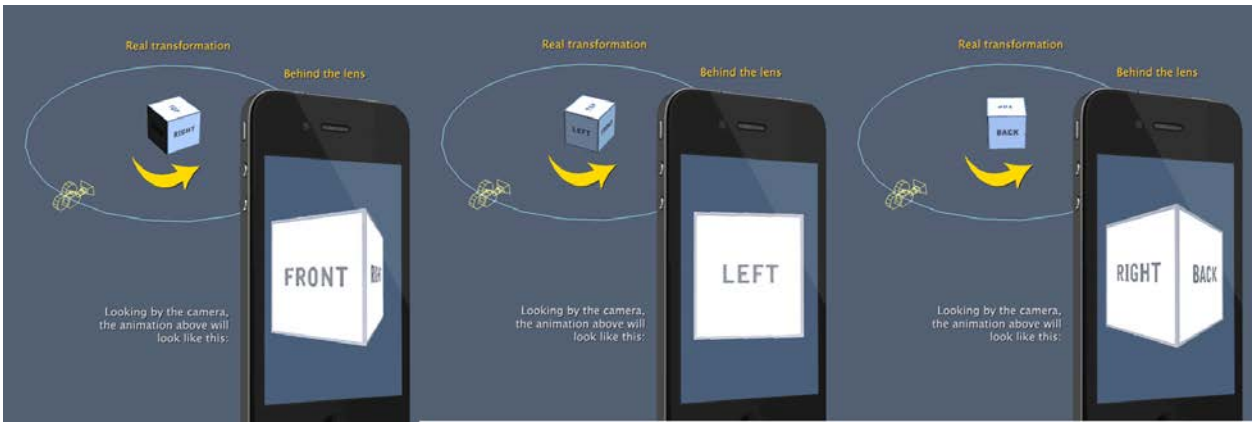
With this, we revisit camera manipulation.

As we saw before, you may think about moving the camera around to take pictures from different points of view. However, in OpenGL the camera is actually fixed, so you have to implement that by moving everything around with a couple of outer matrices. These pictures below illustrate the difference between the two.

You want to revolve the camera around a box while looking at the box:



But OpenGL cannot really do that. So, you keep the camera fixed and rotate the box instead to get the same effect:



The two create essentially the same images.

Let's assume that the box is at the origin of the world (where the wall is). In the first image where we revolve the camera, this is how we would transform the camera if we could:

$M_R.M_T.p_{camera}$

Try to understand this ordering with what we just learned about combining transformations. Let's think about it in the local-to-world order. First we translate the camera and move it away from the box while facing the box. Then we rotate it in the world coordinates. This has the effect of rotating the translation as well, since we are sticking M_R between M_T and the world (the wall). Note that as a result the camera keeps looking at the box.

(If we think about it in the world-to-local order, we first rotate the camera while it coincides with the box, then we move the camera backwards so that it still faces the box.)

In this example, we did not move anything in the world and only moved the camera.

If we assume that camera is the center of the world instead (the wall), we would have to move the rest of the world to get the same effect. We can do this by using the inverse transformation. Instead of transforming the camera, we can transform everything else with the inverse of what we would transform the camera with:

$(M_R.M_T)^{-1}p_{box}$

This means that we want to transform the box with the inverse of what we would transform the camera with. If we do the math:

$M_T^{-1}.\ M_R^{-1}.p_{box}$

The inverse of M_T and M_R are other translation and rotation matrices, let's say M_T' and M_R'

M_T'. M_R '.p_{box}

Which means that (world-to-local order) we first move the box away from the camera, and rotate the box in place locally. This is exactly what is happening in the second picture. (Or we can think of it in the local-to-world order, which would mean that when the box is coinciding with the camera we turn the box, and then we move the box away from the camera).

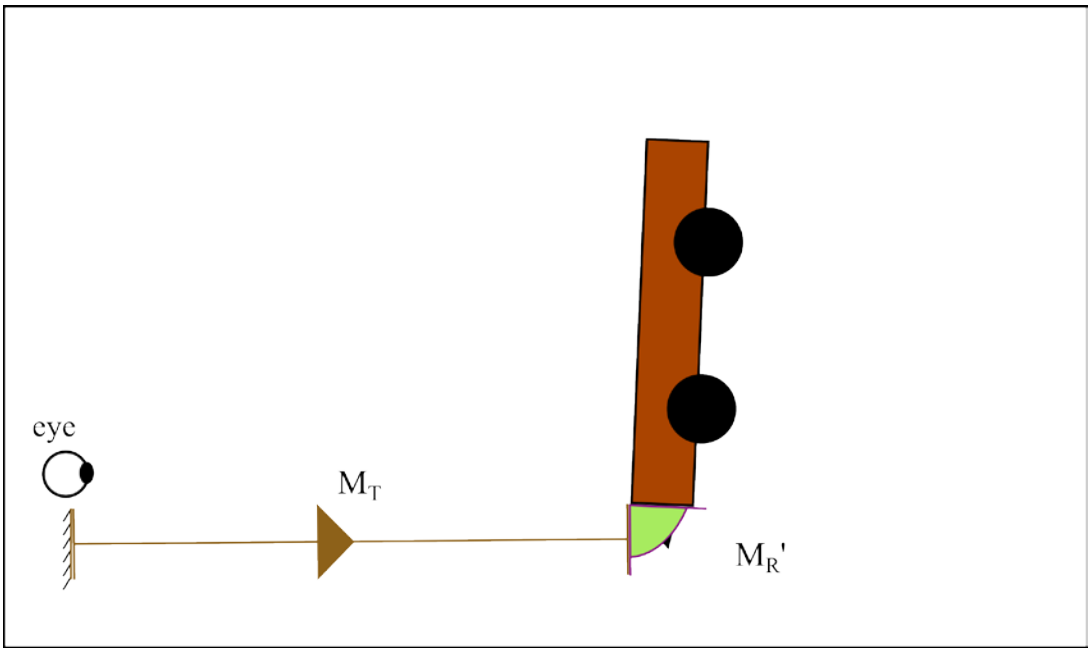
We went through all this trouble to understand that manipulating the camera is implemented as inversely manipulating all the objects in the world. To do this, we have to pre-multiply the modelview matrices used to draw each object with the current camera matrix. As we said, when you write OpenGL code with a series of transformations, you always postmultiply the current matrix. And if you are following code from top to bottom, you have to think of it as world-to-local ordering. This means that you have to put your camera matrix code before any other transformation code, so that it transforms everything else in the world. As we showed on Page 21, we do this by setting the camera matrix before calling transformations of each object. Here is that code just as a reminder:

```
locateCamera()        //sets the current value of the matrix to Mv
glMultMatrix(Mwc)     //current value of the matrix became Mv.Mwc
drawTheCart()         //the cart is drawn as Mv.Mwc.p
locateCamera()        //sets the current value of the matrix to Mv
glMultMatrix(Mwb)     //current value of the matrix became Mv.Mwb
drawTheBox()          //the box is drawn as Mv.Mwb.p
```

As you can see, before drawing the cart, we wanted to move it to its place in the world with the matrix Mwc. However, before that we had to call the camera function that premultiplies the matrix with the inverse camera transformation, like we explained above.

Intuitive Understanding of Camera Manipulation

It helps to visualize camera operations with our wall analogy like this:



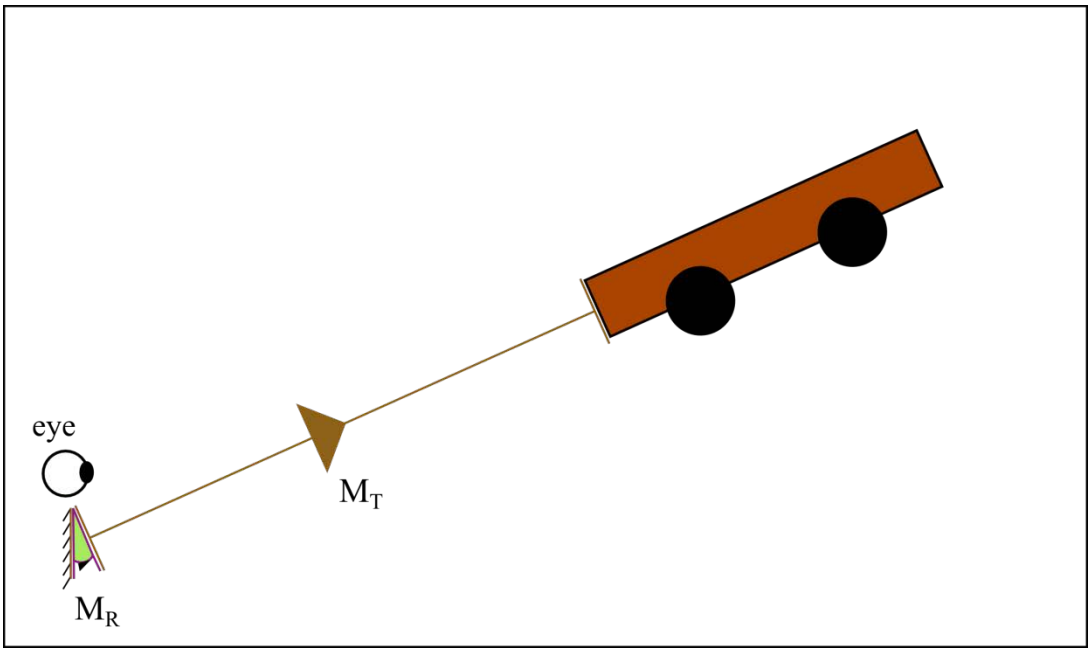
That is, we are looking at the cart (or whatever else is in the world) and we want to move and turn it around with a sequence of transformations so that we can take a look at it the way we want. In this example, we can have a locateCamera() function like this:

```
GLdouble camTy, camTz, camAngle;

void locateCamera() {

      glMatrixMode(GL_MODELVIEW_MATRIX);

      glLoadIdentity();

      glTranslate2f(0, -camTy, -camTz); //it's ok, it's just some MT

      glRotate3f(-camAngle, 1, 0, 0);

}

//Q: explain the choice of axes
```

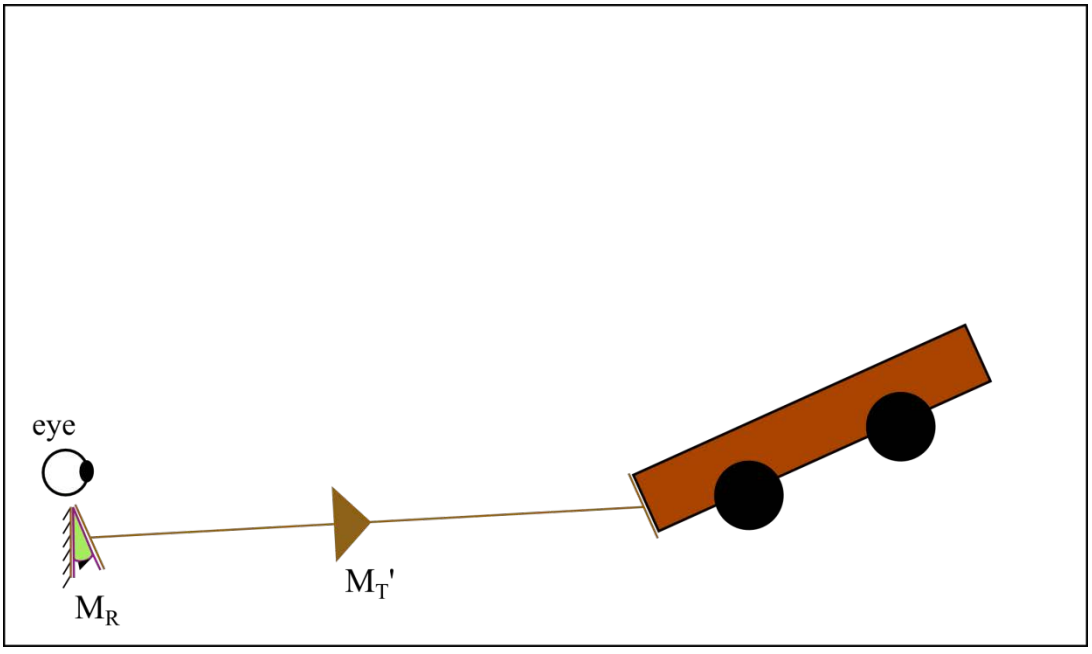
If we call this before drawing everything, it uses the desired values for the location and the orientation angle of the "camera" with respect to our scene. We invert them because we are actually moving everything, not the camera. By changing these global variables in our keyboard or update funcitons for example, we can get the effect of moving the camera.

Note that this camera function is constrained in the sense that it rotates the world while the camera is fixed. Therefore, the camera is actually revolving around the world while looking at a location in the scene. The location that it is looking towards is changed by moving the camera.

As you can see, the order and the kinds of transformations that we use for our camera determine the nature of its motion. For example, we could not have a FPS game effect with the previous camera example. Let's see how we could implement a FPS kind of camera:



Or a better state of it would be:



Note that M_T' is a pure translation just like M_T, because it displaces and maintains orientation.

As you can see, since the very first world transformation here is rotation, it gives the effect of the user turning the head to look around. Then, the only translation after that ensures that the user's head is in

the same relative position with respect to the world. Here is how we could change our previous code to a FPS-like camera.

```
GLdouble camTy, camTz, camAngle;

void locateCamera() {

      glMatrixMode(GL_MODELVIEW_MATRIX);

      glLoadIdentity();

      //simply swapped the two lines below

      glRotate3f(-camAngle, 1, 0, 0);

      glTranslate2f(0, -camTy, -camTz);

}
```

Of course, in a real implementation translation and rotation would be in all three axes.

Therefore you have to choose the behavior of your camera carefully, choose the order of translations and rotations that would locate your camera the way you want, and apply them in your locateCamera() function that you call before any modelview transformations that affect the objects in your scene.

Next up: demoing what we learned in code.

Shading, Lights and Materials

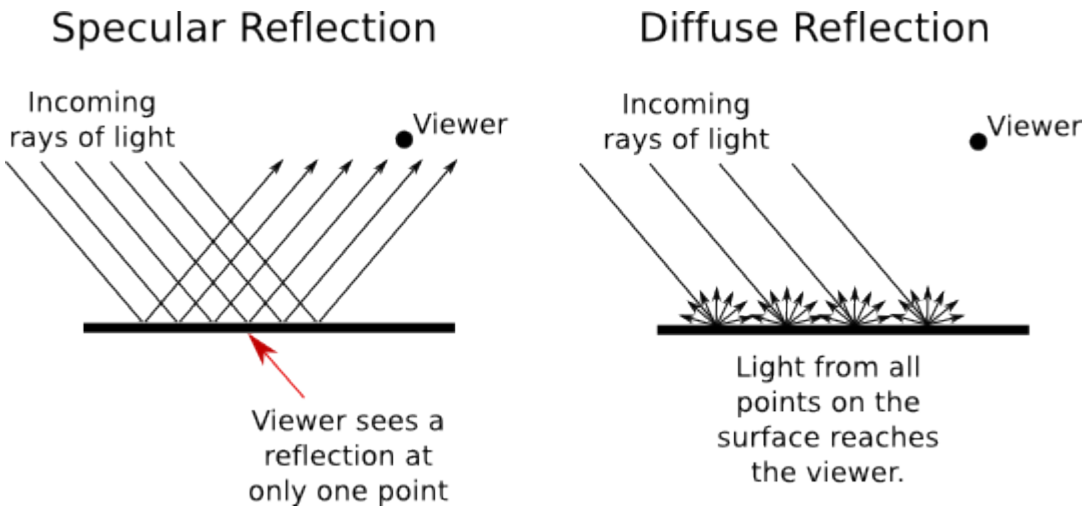
Things that we drew so far with OpenGL were not very realistic-looking. From here on we well learn how to draw geometries in OpenGL so that they look better.

In our previous labs, we saw that if we have a triangle and set its vertices to different colors, OpenGL smoothly transitions between the colors by interpolating the colors.

Normally, the colors that we see on the objects around us depend on the material of the objects (paint, shininess, etc.) as well as the lighting of the environment (sunlight, florescent light, etc.). Similarly, OpenGL can set the vertex colors automatically so that the geometric objects that you draw look realistic.

Lighting from a conceptual level

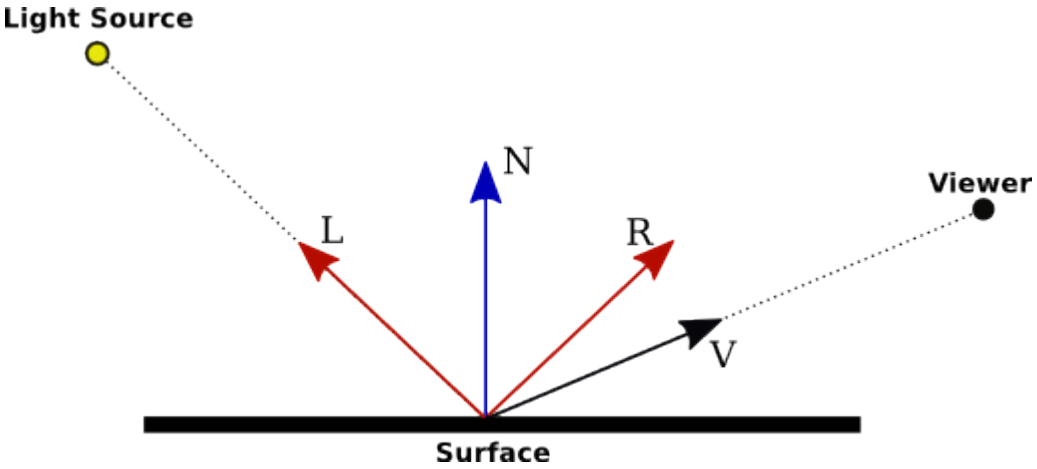
Here is how lighting works in real life.



Lights rays originate from light sources, bounce from surfaces, and then arrive at our eyes to help us see. The light itself has certain intensity and color, the surface has material properties such as color, shininess, etc., and the color and intensity of light that reaches our eye depend on these factors. In addition, the orientation of the surface also determines the characteristics of the light that bounces off of it. For example, if the light rays arrive at the surface with a right angle, the object appears well-lit. If the angle is close to tangent the surface is less-lit.

Lighting in OpenGL

OpenGL approximately simulates the physical attributes of lights as well as materials and geometry of surfaces to determine the color that the surface appears to have.



For example, if the viewer was along the ray R, then the surface point would look slightly brighter because it is how most rays from the light source toward this point would bounce.

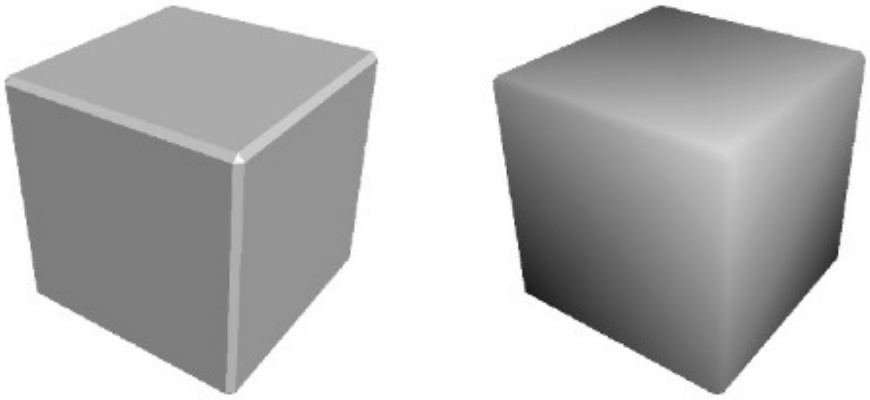
Using attributes related to the lights and the surface parameters, OpenGL calculates a color for each vertex on a polygon, and interpolates these values to fill the polygon.

The attributes that determine the lighting of a vertex are parameters of lights, parameters of the surface material and the surface normal at the vertex. We will see the parameters of lights and materials later. First, let's see why we need to define surface normal vectors for appropriate lighting.

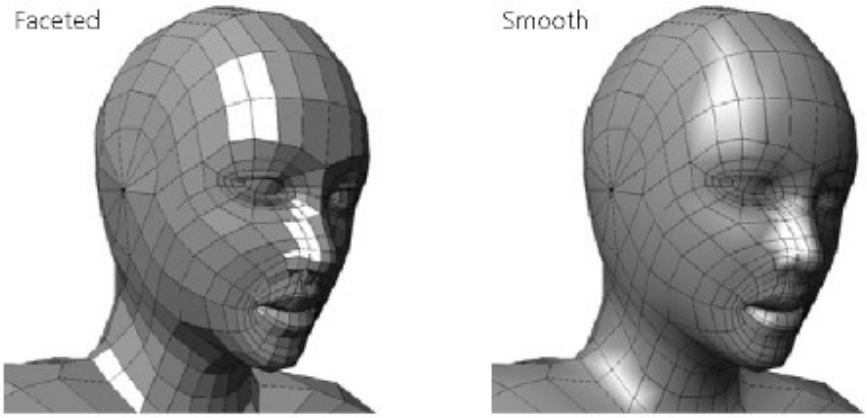
Surface normal vectors

Surface normal vector at a point on a surface is a vector that is perpendicular to the tangent of the surface at that point. As we see in the image above, the incoming light ray and the light ray that bounces off the point on the surface have the same angle with the normal. OpenGL needs this normal vector on a vertex in order to calculate how the ray will bounce. While for most polygonal shapes we can calculate what the normal should be, normals are often used in slightly inaccurate ways to create different shading effects. Here is an example:

Faceted polygons are appropriate for geometric shapes like dice.

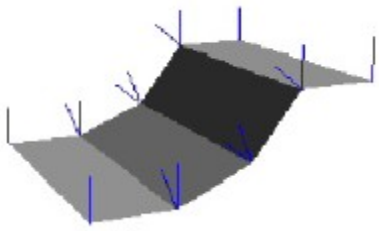


Smooth polygons are appropriate for organic shapes like faces.

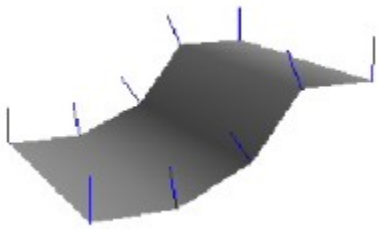


The two images on the right and the left are rendered using the same vertices that define quads and triangles. The ones on the left use normals separately for every face, as they should be. We can clearly see how the quads and triangles in the image are discrete from each other. Conversely, on the right we see the same vertices rendered with normals that are averaged with the neighboring faces. As you can see, the rendered image appears round even if it is not round. This is one way of using normals to make the model appear realistic.

Here is a representation of normal vectors on vertices of a large polygon.



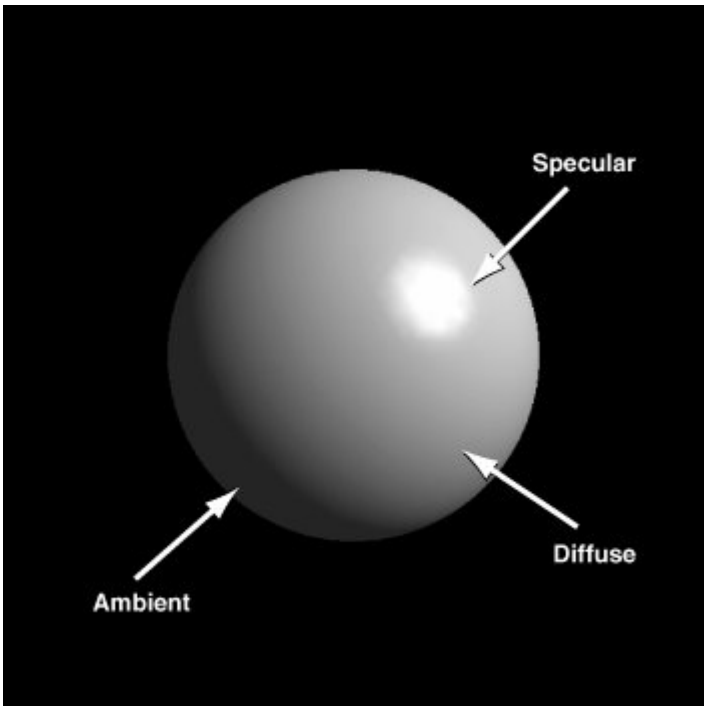
Here is the same polygon, with the normals of the neighboring vertices are averaged to be the same:

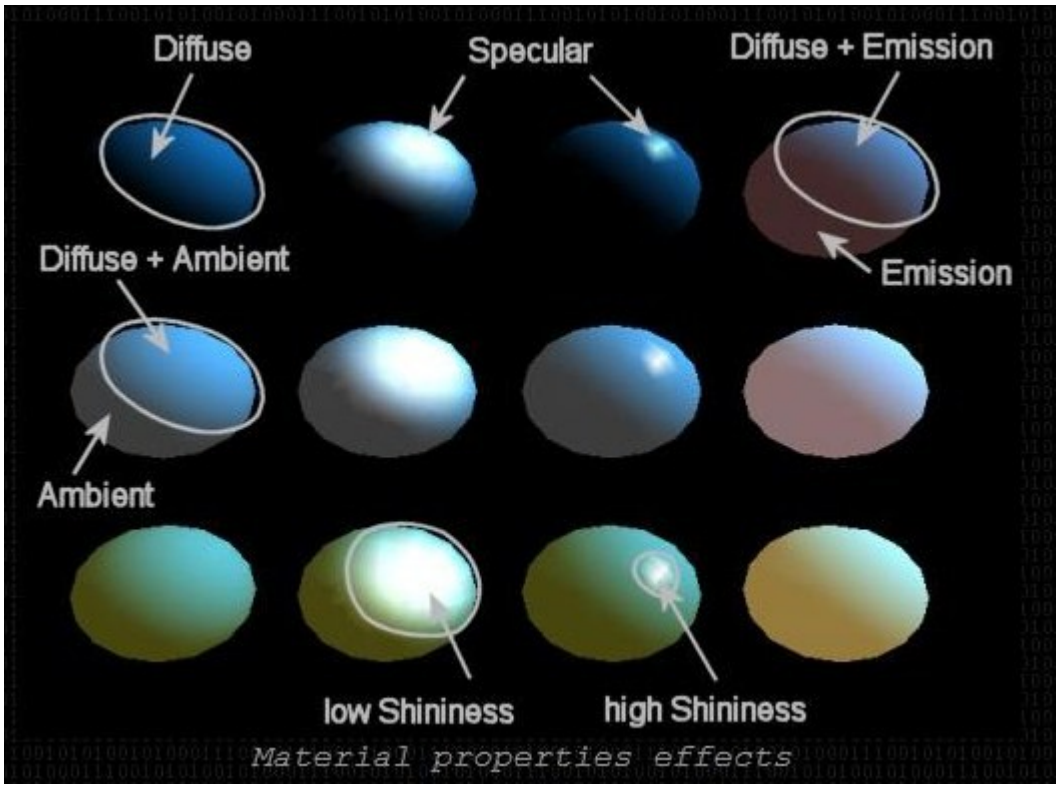


Since there are no abrupt normal vector changes, the neighboring edges of the quads appear to have the same color, which gives the round look.

Light and material attributes

According to OpenGL, the light rays that arrive at our eye come through four light channels: ambient, diffuse, specular and emission. Below are what we mean by each:





(from http://www.sjbaker.org/steve/omniv/opengl_lighting.html)

AMBIENT - light that comes from all directions equally and is scattered in all directions equally by the polygons in your scene. This isn't quite true of the real world - but it's a good first approximation for

light that comes pretty much uniformly from the sky and arrives onto a surface by bouncing off so many other surfaces that it might as well be uniform.

DIFFUSE - light that comes from a particular point source (like the Sun) and hits surfaces with an intensity that depends on whether they face towards the light or away from it. However, once the light radiates from the surface, it does so equally in all directions. It is diffuse lighting that best defines the shape of 3D objects.

SPECULAR - as with diffuse lighting, the light comes from a point souce, but with specular lighting, it is reflected more in the manner of a mirror where most of the light bounces off in a particular direction defined by the surface shape. Specular lighting is what produces the shiney highlights and helps us to distinguish between flat, dull surfaces such as plaster and shiney surfaces like polished plastics and metals.

EMISSION - in this case, the light is actually emitted by the polygon - equally in all directions.

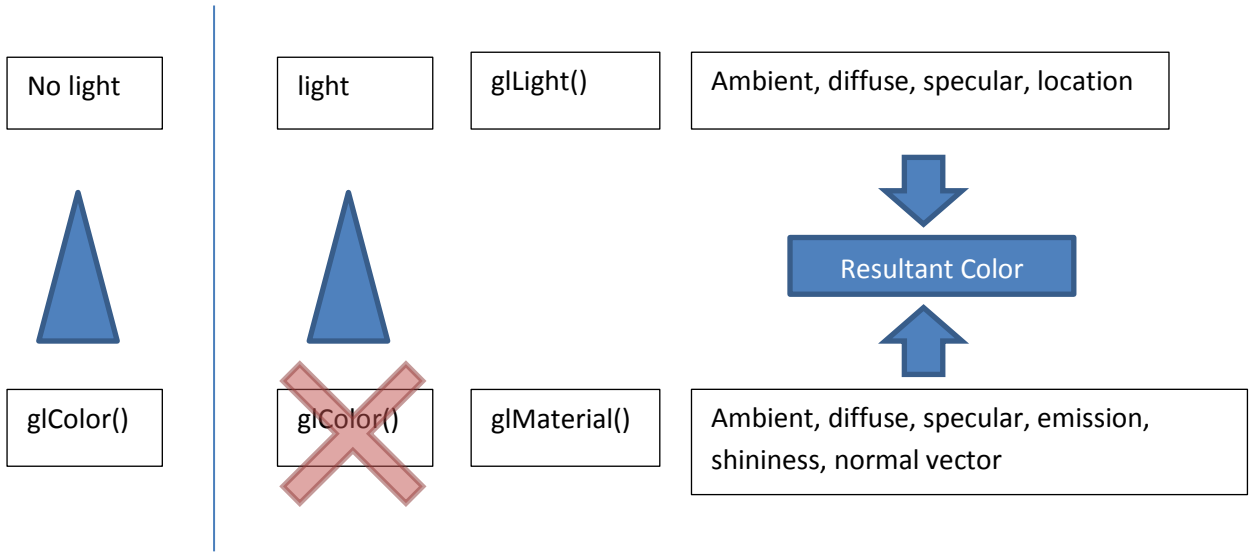
Lights in OpenGL have ambient, diffuse and specular components. Surface materials in OpenGL have ambient, diffuse, specular and emission components. All of these are combined with the surface normal to give us the final color of a vertex on a surface.

Implementing Lighting in OpenGL

Here are some excellent resources on OpenGL lighting:

http://glprogramming.com/red/chapter05.html

http://www.sjbaker.org/steve/omniv/opengl_lighting.html



In OpenGL, you enable lighting with glEnable(GL_LIGHTING). When you enable lighting, your glColor() calls are ignored and your objects appear black unless you create a light and set material properties.

Here is sample code on how you set up and enable a light

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

By setting different components of the light (ambient, diffuse, specular, position, etc.) you define a light. You enable the light with glEnable(). You can change these properties of the light later. For example, you can move the light by calling glLight with GL_POSITION again later.

We also need to set up the material properties of our surfaces for lighting to work. Here is sample code for how to do that:

```
GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 };
GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat low_shininess[] = { 5.0 };
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

TODO link to the sphere teapot lighting example from korea

You do these glMaterial calls instead of glColor calls.

This way, we set up lights and define material properties. We also need our normal vectors for every vertex. We do it using glNormal as follows:

```
glBegin(GL_TRIANGLES);
glNormal3d(0.0, 0.0, 1.0);
glVertex3d(1.0, 0.0, 0.0);
glVertex3d(0.0, 1.0, 0.0);
glVertex3d(-1.0, 0.0, 0.0);
glEnd();
```

With glNormal, we set the current normal vector that will be used with the vertices that follow. If normals for the vertices are different, we have to keep changing them. For example:

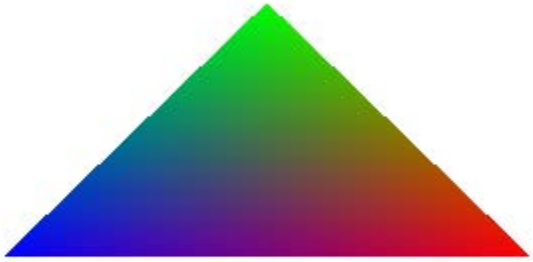
```
glBegin(GL_TRIANGLES);
glNormal3d(0.0, 0.0, 1.0);
glVertex3d(1.0, 0.0, 0.0);
glNormal3d(0.0, 1.0, 0.0);
glVertex3d(0.0, 1.0, 0.0);
glNormal3d(1.0, 0.0, 0.0);
glVertex3d(-1.0, 0.0, 0.0);
glEnd();
```

You can use functions such as glutSolidSphere that draws a sphere with appropriate normals to test your lighting code.

Next time, we will learn about textures.

Texture Mapping

Up till now, we defined colors of our geometric objects only on the vertices. We did this either by setting the color ourselves or letting OpenGL lighting determine the color. After colors of vertices were determined, OpenGL interpolated them to fill the rest of the geometry.



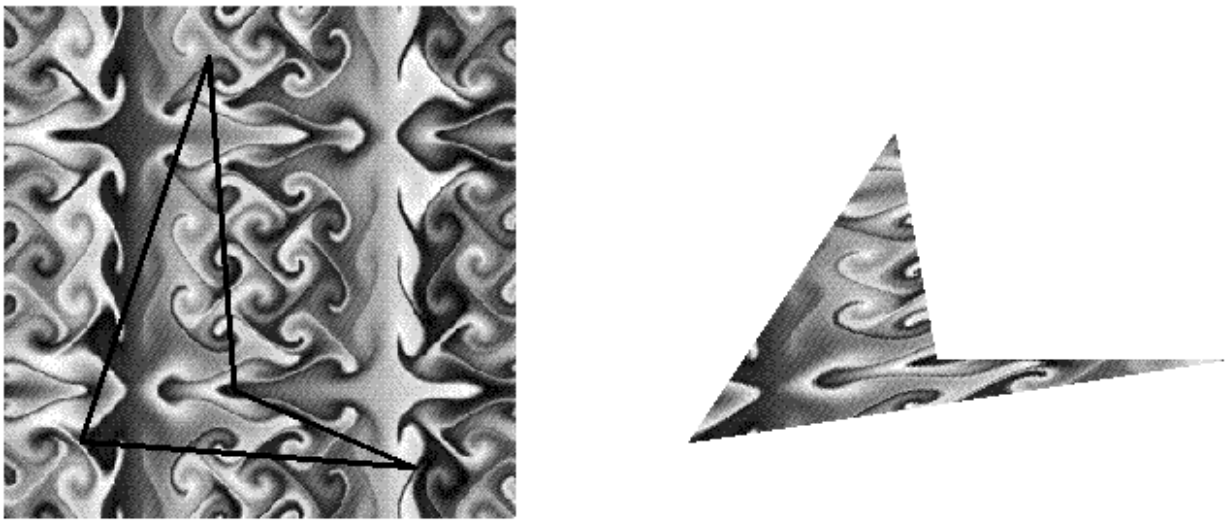
Therefore, we did not have direct control over what color the middle of a triangle (or a quad) would have. OpenGL calculated it using the colors of the vertices. Let's say, if we wanted the middle of the triangle to have a red circle drawn on it, we could not do it with what we have learned so far.

To determine what color every point on a triangle is, we use textures. Textures are bitmap images mapped onto a geometrical object. Using texture mapping, we can draw the red circle inside the triangle. You can imagine textures in terms of covering the geometry by stretching an image on it.

There are different kinds of textures, including 1D, 2D and 3D textures. We will only focus on 2D textures.

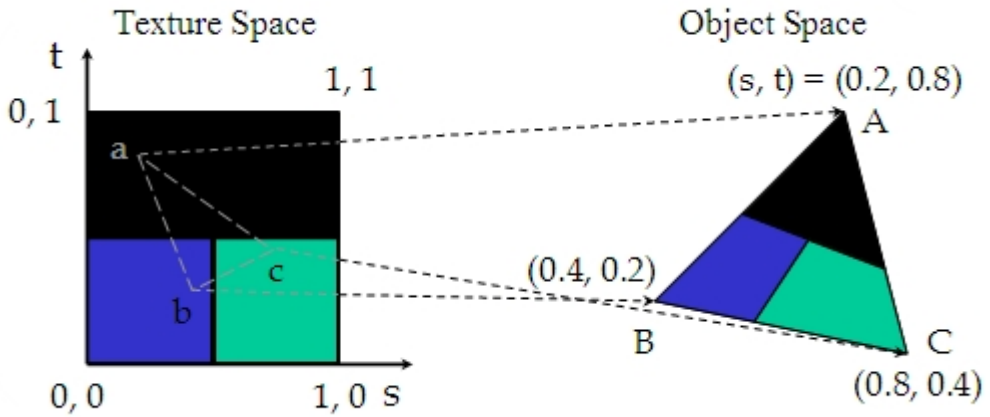
How Textures Work

Here is how 2D textures work. You supply a 2D bitmap image, and create a mapping of the vertices of your geometry on the image. You can imagine the rest as cutting the mapped area from the image, resizing or stretching it to bring it to be the same shape as the geometry, and sticking it on top of the geometry. Here is an example:



On the left there is a 2D image with a certain pattern. On the right is a geometrical shape in 3D that is composed of two triangles. The points on the geometrical shape are mapped on the image on the left as shown with the black lines. Then, the area in the image that is inside the mapped vertices is used to determine the colors of all the points on the geometrical shape. This gives a stretching effect depending on the mapping.

Here is how this works with a little more detail:

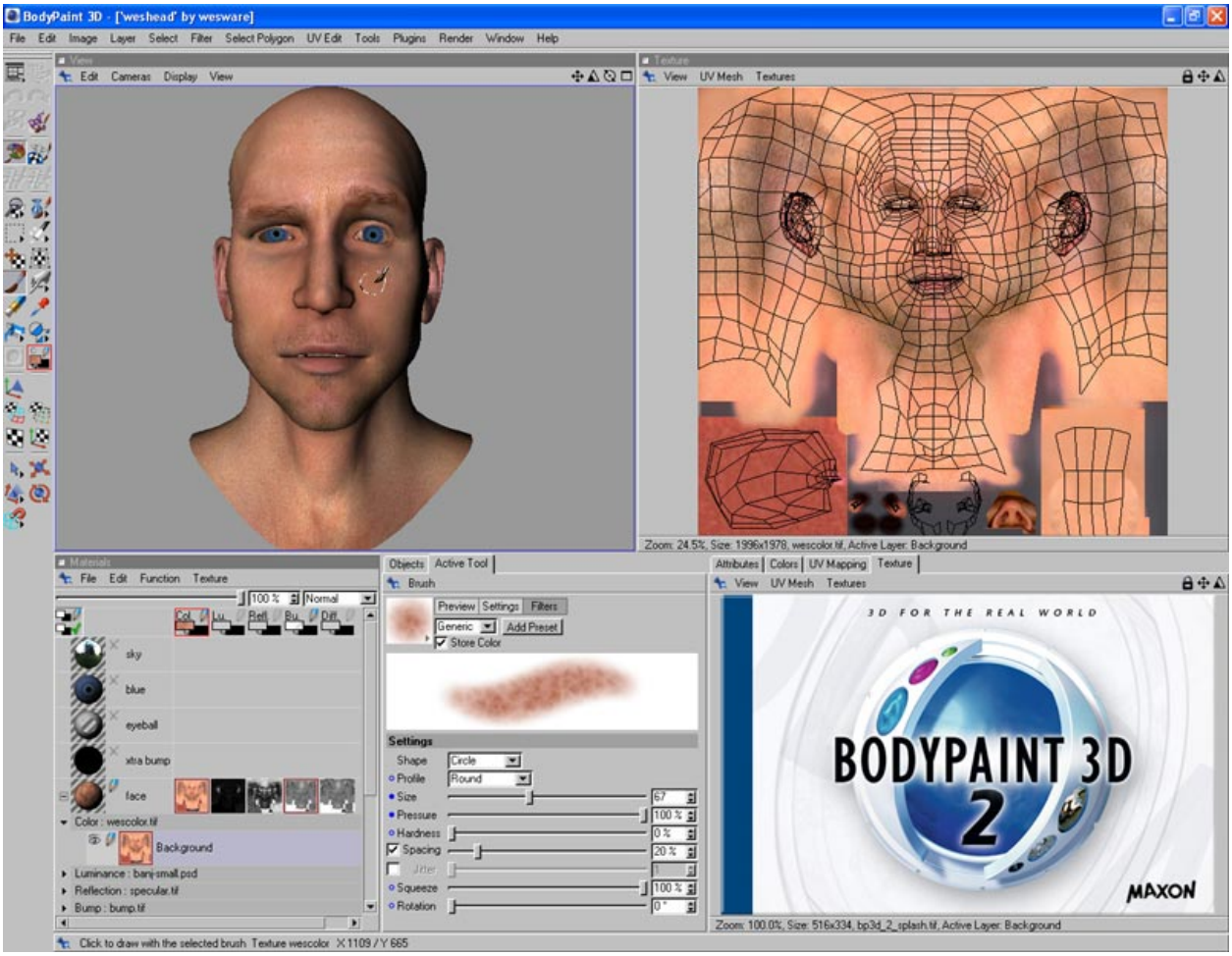


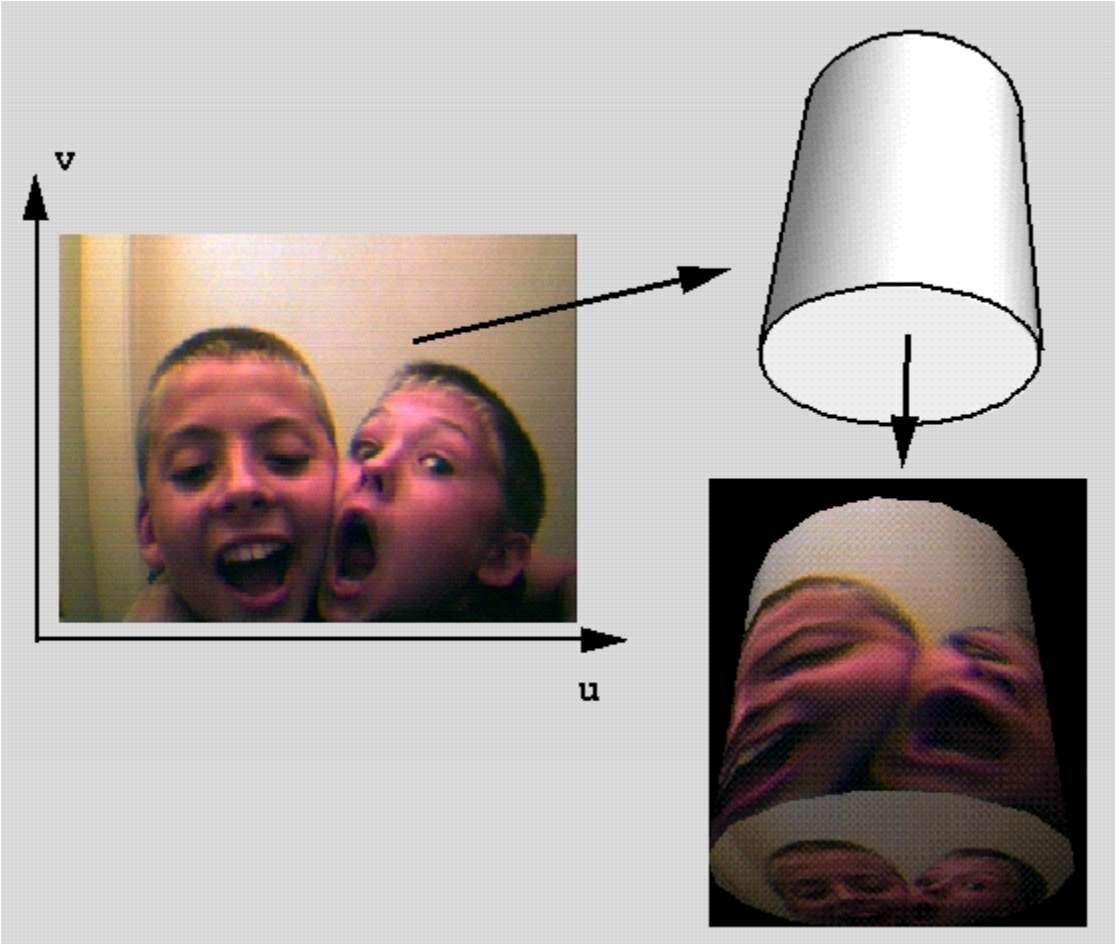
The image on the left contains black, blue and green regions. The width and height of the image are expressed in the interval [0,1] as normalized coordinates. In this coordinate system, we pick three points: a = (0.2, 0.8), b = (0.4, 0.2), and c = (0.8, 0.4).

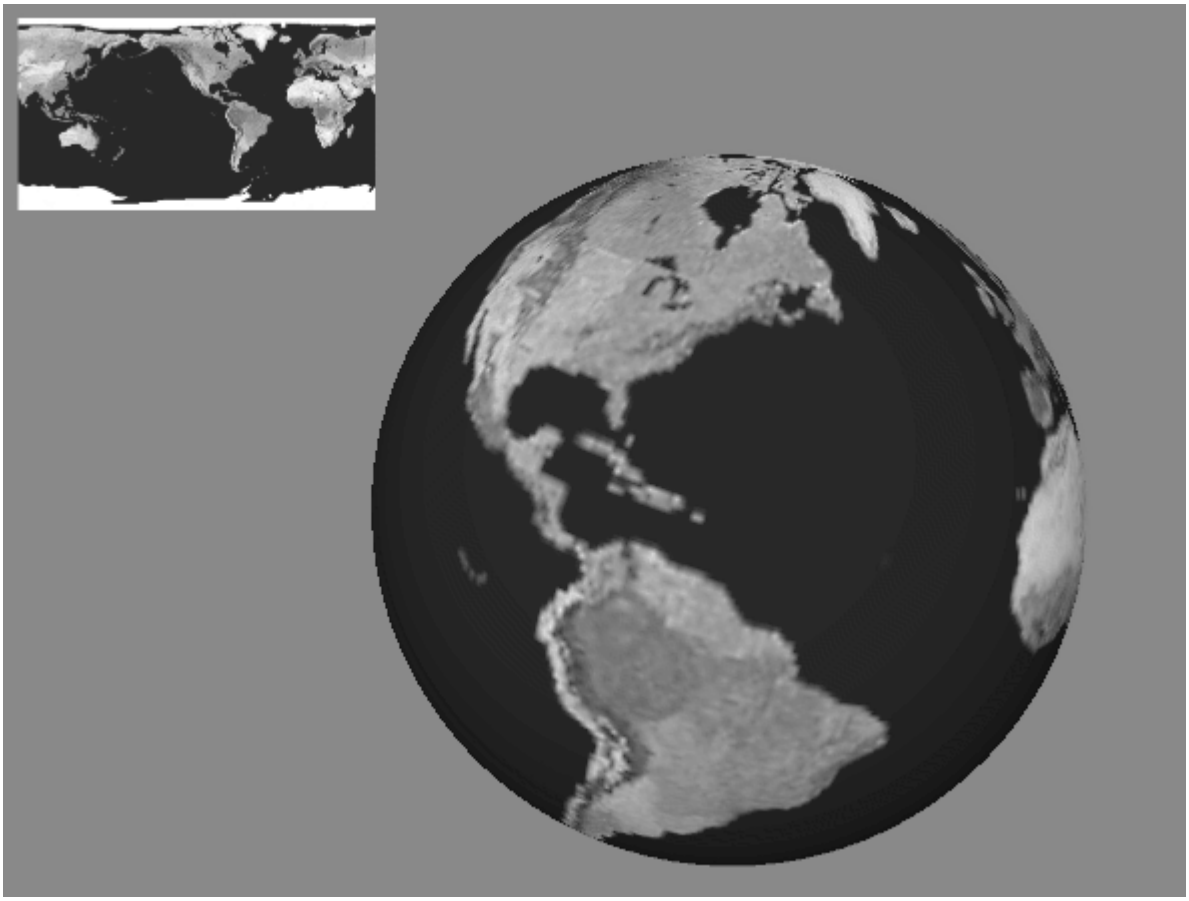
On the right we have a triangle in 3D with vertices A, B and C. We map the points a, b and c from normalized texture coordinates to these 3D points respectively. As a result, the image within a, b and c

is reshaped to get the shape of the 3D triangle and is used to determine the color of each point on the triangle.

Here are some sample uses of texture mapping:

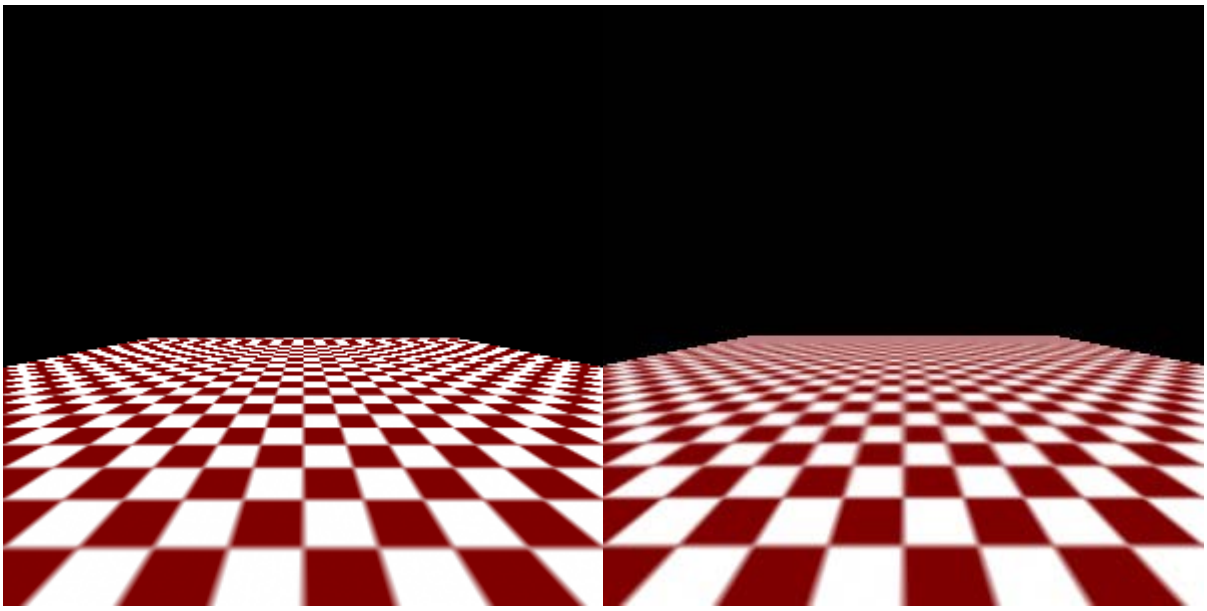



v
u



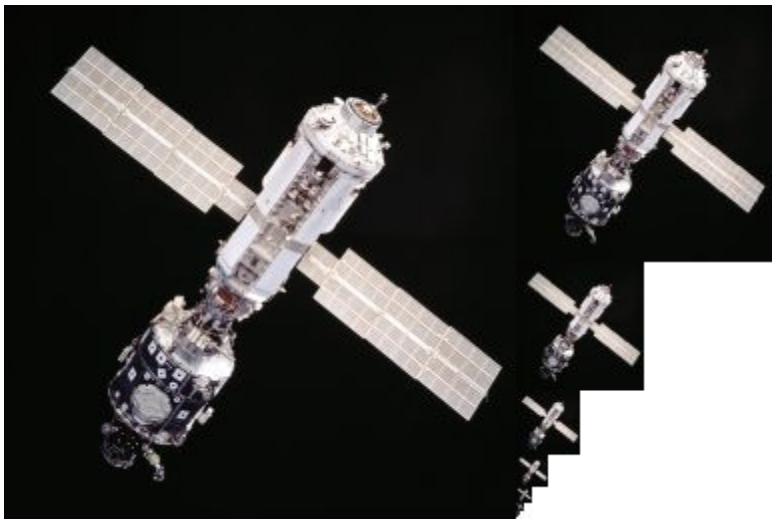
Filtering

When the rendering system wants to render a point on the 3D triangle, it uses this mapping to determine where exactly in the image it should read the color from. Since the image is a bitmap that is just a grid of colors, the resolution of the image may not always be ideal and undesired side effects may occur.

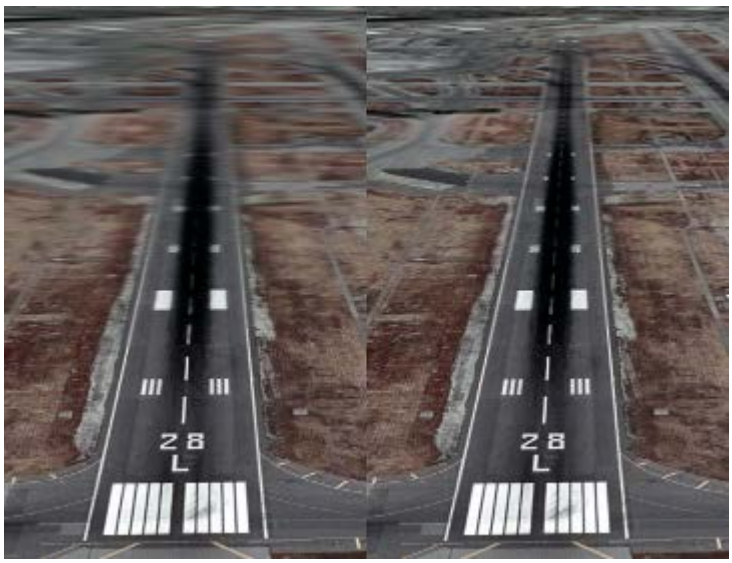


OpenGL simply picks a color from the texture image for each pixel that is used to draw the ground here. The further areas should appear to us as a mixture of white and red. However, OpenGL reads either white or red from the image, and uses them. As a result, all the pixels of the image on the left are either white or red.

However, the image on the right appears more blurry as you go further, which is what we would expect. This is achieved with a technique called **mipmapping**. Instead of using the high-resolution texture image, OpenGL automatically creates a number of scaled down copies of the texture image. It uses smaller copies of the texture for geometry that is further from the eye. This scaling results in the necessary blurring, and OpenGL uses the blurred pixels instead of the original sharp pixels.



Another related technique is **anisotropic filtering**. Sometimes, rather than simply going further from the camera, textures are located on geometry that has a narrow angle towards the camera.



To get better texture mapping in such situations, rather than blurring the image evenly by scaling it down uniformly, we need to scale it in one direction only. Here is an example:

Depending on the viewing angle, OpenGL chooses the appropriate texture to sample from to get the best rendering results.

Texture Mapping in OpenGL

To use texture mapping in OpenGL, here is a rough sketch of things that need to be done:

- Enable textures
- Generate texture ids
- Bind one of the texture ids
- Load a texture image into the active texture id
- When drawing your geometry, give which texture coordinates should be mapped to each vertex

Here is roughly how they are done:

Enable textures

This is simple:

```
glEnable ( GL_TEXTURE_2D );
```

You can enable and disable textures for different geometries that you draw.

Generate texture names

When you have a lot of geometries with different textures on them, OpenGL has to keep track of many different textures and let you use them on different geometries as you want. This is achieved through texture ids. In the GPU, OpenGL stores a texture for each texture id, and lets you recall any texture using its id.

To be able to use these texture ids, you first have to tell OpenGL to reserve some texture ids for you. You do this by

```
GLuint texture_id[n];
glGenTextures (n, texture_id);
```

This creates n texture ids and places them in the array texture_id. Later you can use these ids to load textures into them or use the textures loaded on them.

Bind one of the texture ids

When you load a texture to the GPU, you have to load it to one of the texture ids that you created. For this, you first have to choose which texture id that you will use. That is, you “bind” the id that you will use to make it the active texture id in the OpenGL state. You do this by

```
glBindTexture ( GL_TEXTURE_2D, texture_id[0] );
```

This makes the first created texture id to be the active 2D texture id.

Load a texture image into the active texture id

You first have to create the texture image in the RAM somehow. For this, you usually use a library routine to load the texture from a file. You can also create arrays of bytes and create the texture programmatically. Once you have the data in RAM, you use glTexImage2D() to load it to the GPU under the active texture id. You can also use gluBuild2DMipmaps() to automatically create a mipmapped version of the texture and load it to the GPU. Here is some sample usage.

```
gluBuild2DMipmaps ( GL_TEXTURE_2D, internal_format, p->info.width, p->info.height, p->info.tgaColourType, GL_UNSIGNED_BYTE, p->data );

glTexImage2D ( GL_TEXTURE_2D, 0, internal_format, p->info.width, p->info.height, 0, p->info.tgaColourType, GL_UNSIGNED_BYTE, p->data );
```

Please see the man pages of these functions, or take a look at these tutorials to learn about managing images:

http://www.nullterminator.net/gltexture.html

http://www.gamedev.net/page/resources/_/technical/opengl/opengl-texture-mapping-an-introduction-r947

http://nehe.gamedev.net/tutorial/texture_mapping/12038/

When drawing your geometry, give which texture coordinates should be mapped to each vertex

This is the last step for using a texture. To tell OpenGL how the texture is mapped on the geometry, you give the normalized image coordinates that each vertex should be mapped to. Here is an example:

```
glBindTexture ( GL_TEXTURE_2D, texture_id[0] );
glBegin ( GL_QUADS );
      glTexCoord2f(0.0f, 0.0f);
      glVertex3f(-1.0f, -1.0f,  1.0f);
      glTexCoord2f(1.0f, 0.0f);
      glVertex3f( 1.0f, -1.0f,  1.0f);
      glTexCoord2f(1.0f, 1.0f);
      glVertex3f( 1.0f,  1.0f,  1.0f);
      glTexCoord2f(0.0f, 1.0f);
      glVertex3f(-1.0f,  1.0f,  1.0f);
glEnd()
```

First, you make sure the texture you want is bound. Then, similar to setting color, you set the current texture coordinate before calling glVertex. By providing the 2D coordinates (st coordinates) for each vertex, you define how the texture is mapped onto the geometry.

Next time, we will see more advanced techniques related to texture mapping.

Textures and colors

Up till now we did not talk about how glColor calls and how texture mapping interact. Colors set in glColor calls and the colors that come from the texture image are multiplied to find the resulting colors. Here is how they are calculated for each pixel:

(R, G, B) = (Rc * Rt, Gc * Gt, Bc * Bt)

Therefore, to ensure that your texture image is displayed the way it is, you have to ensure that glColor(1, 1, 1) is called.

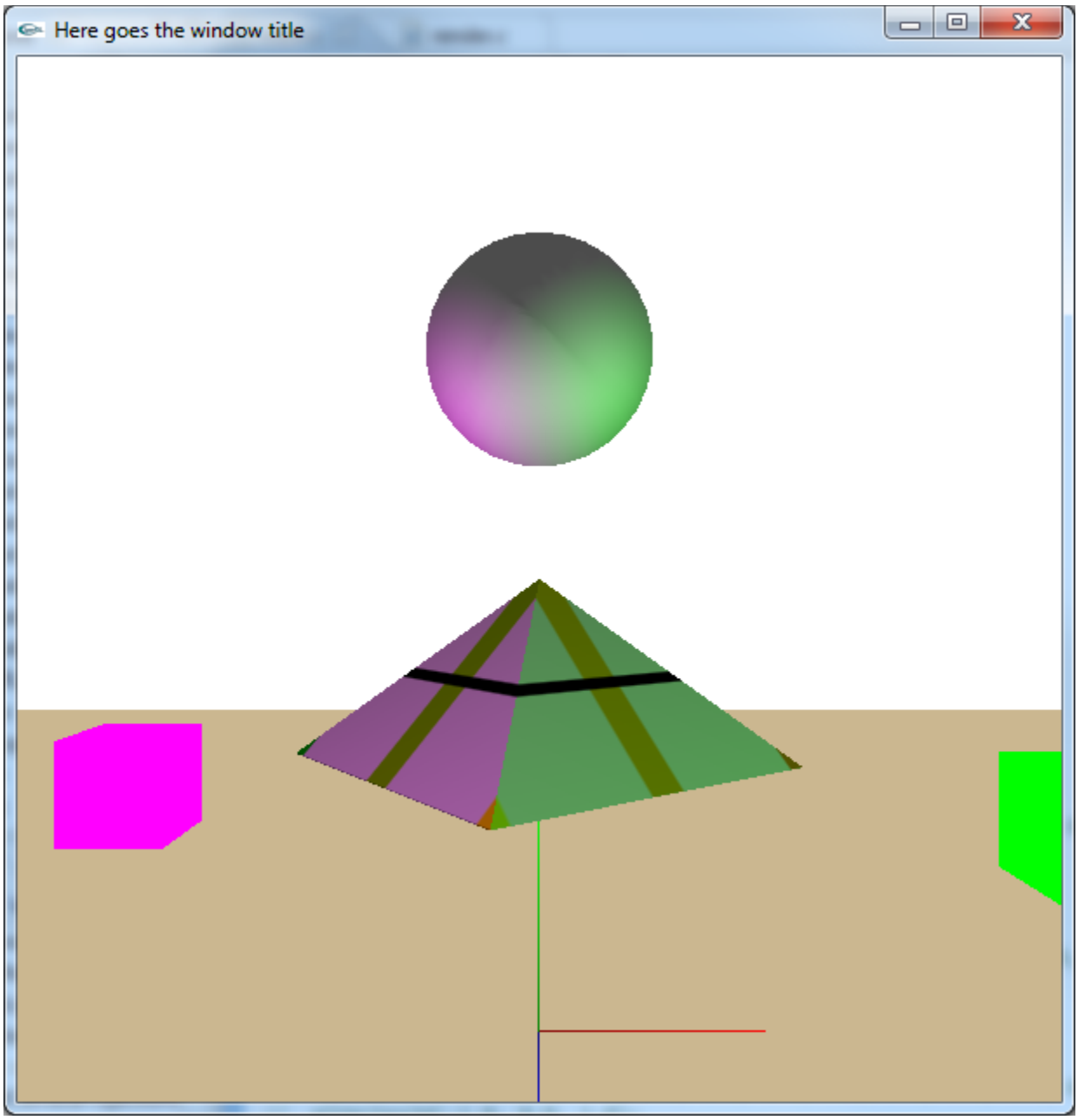
For example, if you call glColor(1, 0, 0), only the red colors in the image will be visible and all the green and blue components of colors will go to black.

Textures and lighting

As we learned before, OpenGL lighting is just a substitute for glColor calls. OpenGL calculates the appropriate color for each vertex according to the light position, vertex position, vertex normal, light parameters and material parameters. Therefore, OpenGL lighting interacts with textures the same way glColor interacts. That is, the color of the texture is multiplied with the color that comes from lighting calculations. The pixels colors are calculated similarly:

(R, G, B) = (Rl * Rt, Gl * Gt, Bl * Bt)

Below you see an example from our previous labs. In this image, both textures and lighting is enabled for the pyramid and we see that the colored lights are shining on the texture:

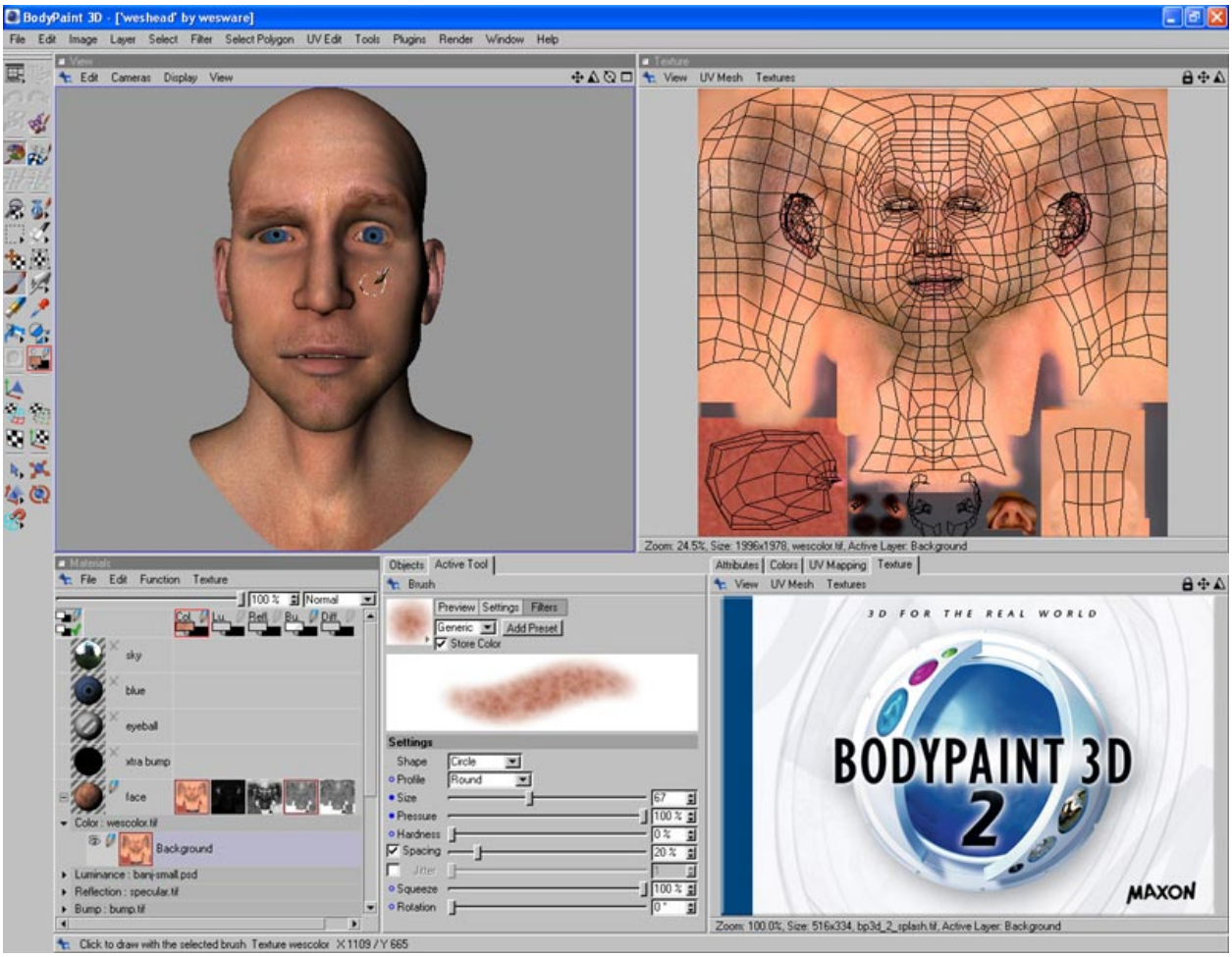


Mixing lighting with textures results in more realistic renderings. Without lighting, textures are simply copied on the geometry and shading effects are lost.

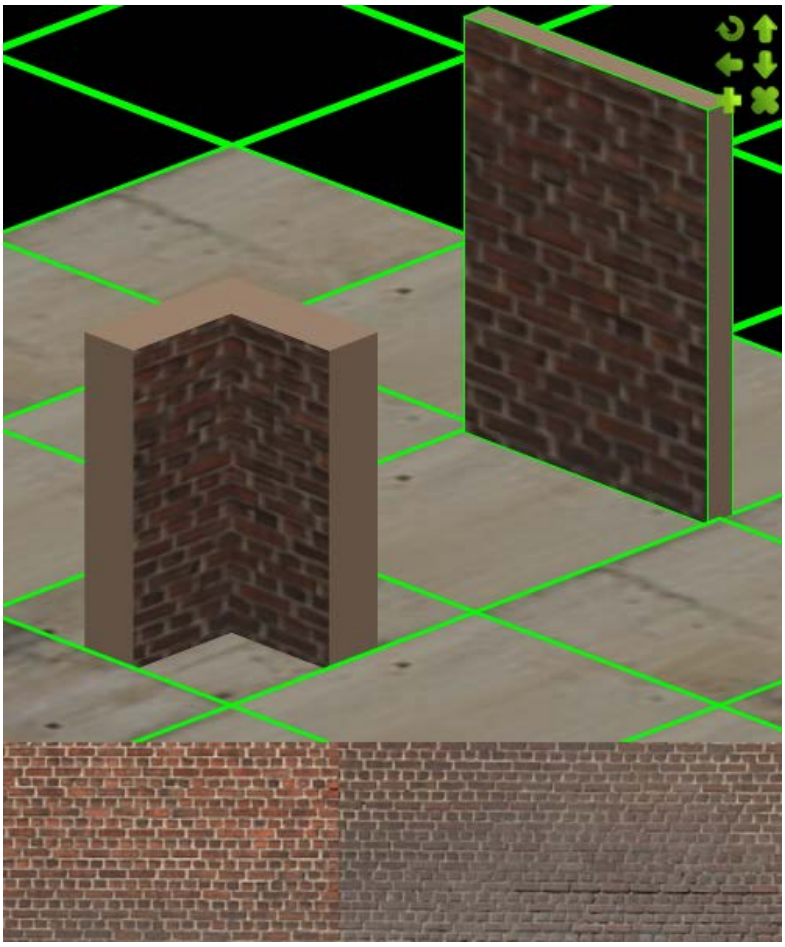
Generating Texture Coordinates

As we saw before, for every vertex in our geometrical model, we have to provide a 2D texture coordinate to map a texture onto the model. Giving them by hand can be a very tedious job.

This can be handled in software packages like the one that we talked about before:



However, sometimes we do not care about how exactly the texture is mapped. Sometimes it is enough to have the model somehow display the texture on it.



Walls are a good example. Maybe we do not care exactly which vertex has which part of the wall texture, but we just want the object to roughly have the texture on it.

For such purposes, we can ask OpenGL to calculate texture coordinates automatically for us. For this we use the glTexGen function to tell OpenGL how it should calculate the texture coordinates.

void glTexGen{ifd}[v](GLenum coord, GLenum pname, TYPEparam)

Coord can be GL_S, GL_T, GL_R, GL_Q.

GL_S and GL_T are the 2D texture coordinates. GL_R and GL_Q are related to perspective-correct texturing. You can read more about them here and here.

Pname and **param** can be:

Pname	Param
GL_TEXTURE_GEN_MODE	GL_OBJECT_LINEAR
	GL_EYE_LINEAR
	GL_SPHERE_MAP
GL_OBJECT_PLANE	Array with plane coefficients
GL_EYE_PLANE	Array with plane coefficients

With this, you can tell OpenGL how exactly it should generate texture coordinates.

GL_OBJECT_LINEAR creates the texture coordinates relative to the object, so that if you rotate the object the texture seems to stay on the object the same. In contrast,

GL_EYE_LINEAR creates them relative to the camera. Therefore, when the object turns, the texture seems to “slide” on it. However, this is not a realistic mapping.

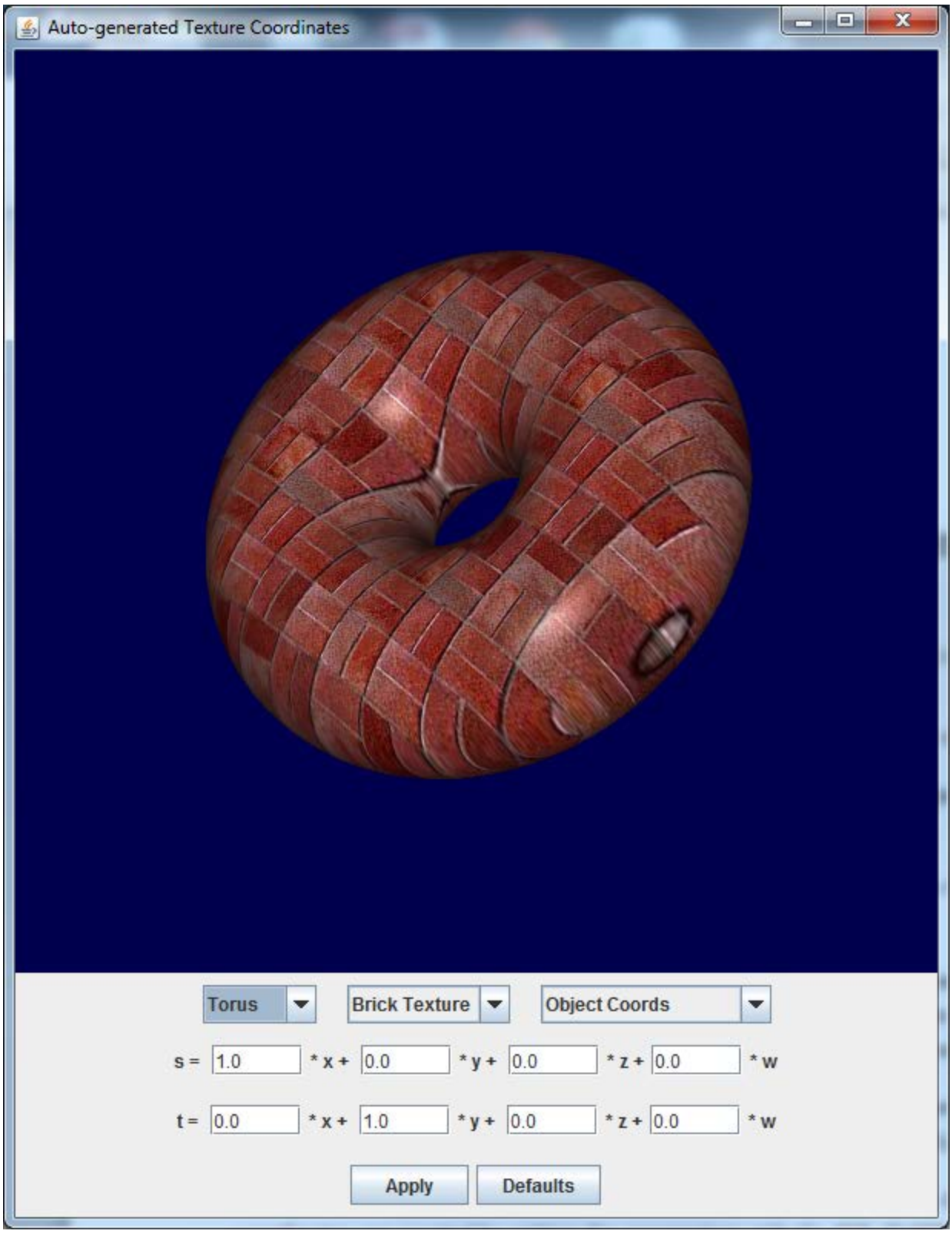
GL_SPHERE_MAP does a special kind of mapping that makes it look like the object is very shiny and the texture is in the environment. This is more realistic to the previous one as it looks like the object is reflecting what is there in the environment. This is also called “environment mapping”.

If you choose GL_OBJECT_LINEAR, you can choose which plane relative to the object that the texture coordinates are created by setting the parameters with GL_OBJECT_PLANE for both GL_S and GL_T.

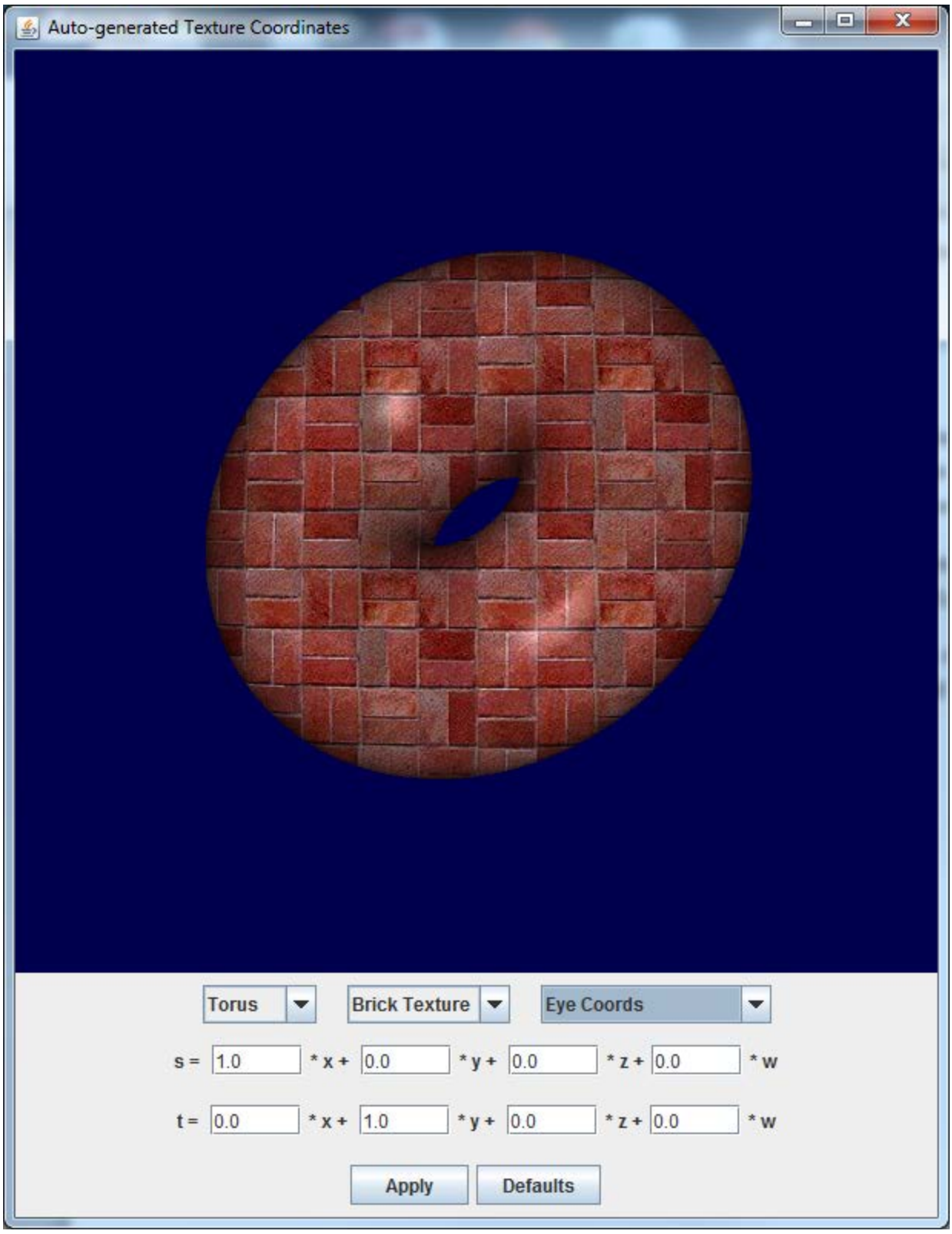
If you choose GL_EYE_LINEAR, you can choose which plane relative to the object that the texture coordinates are created by setting the parameters with GL_EYE_PLANE for both GL_S and GL_T.

Here are some screenshots from the source code here.

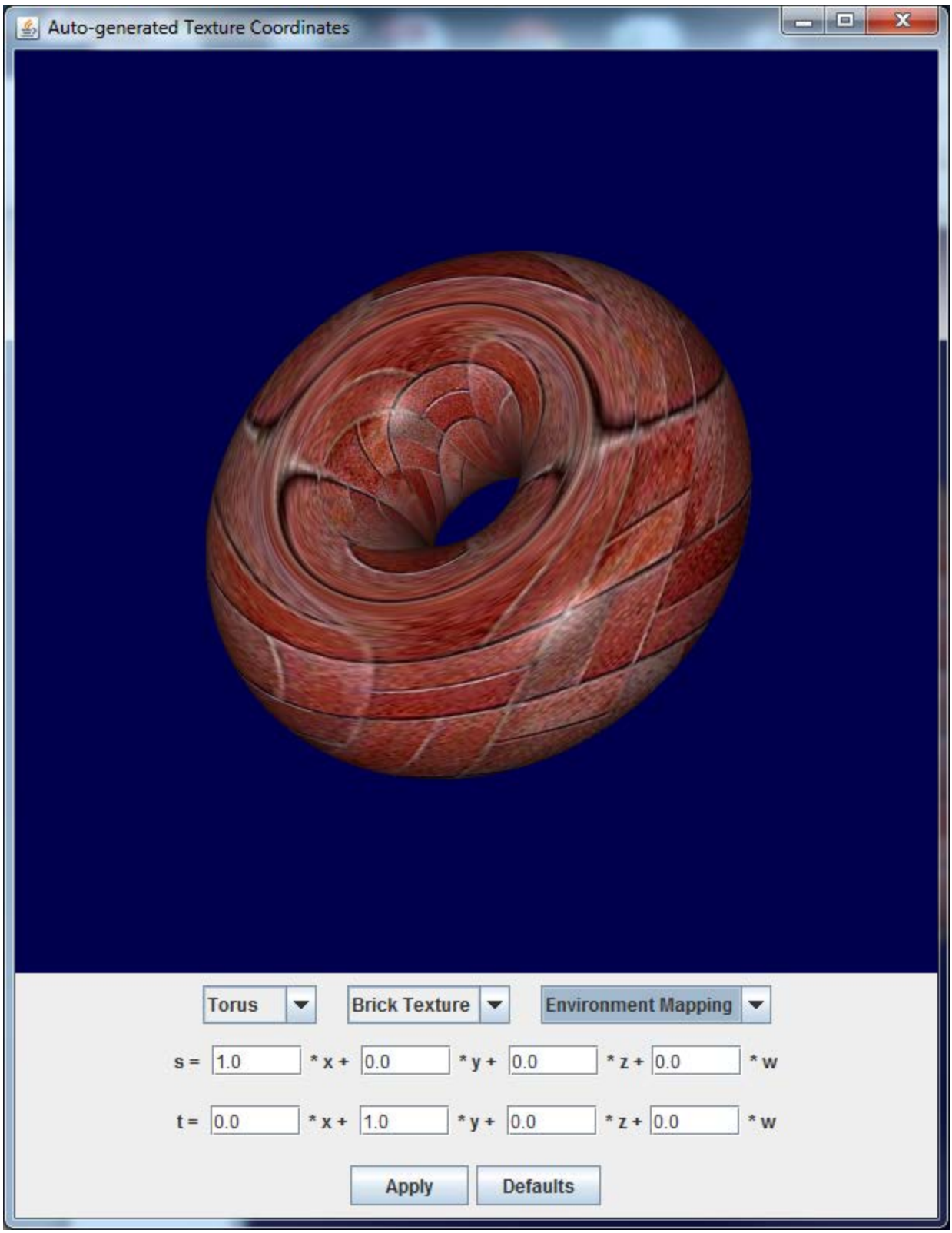
Object coordinates:



Eye coordinates:



Environment mapping:


Auto-generated Texture Coordinates
Torus
Brick Texture
Environment Mapping
s = 1.0 * x + 0.0 * y + 0.0 * z + 0.0 * w
t = 0.0 * x + 1.0 * y + 0.0 * z + 0.0 * w
Apply
Defaults

Up till now we have been teaching practical knowledge about computer graphics. Now we will gloss over general computer graphics concepts that you should be aware of but do not need to know in detail.

Creating 3D meshes

We said that artists create the models in a 3D editor and save it with local coordinates of the model. Let's see a simple demonstration.

BLENDER DEMO

These models have vertices, faces and textures. It would be impossible for us to write down glBegin()/glEnd() code and give the values by hand. There are libraries that do this for us. They read models with their vertices, faces, materials, normals, etc. from a file, put this information in data structures and call OpenGL commands to render the model.

Dealing with Large Models

As you can see, most models used in real graphics applications have many vertices and faces. As a result, every time the scene is rendered, we call glBegin()/glEnd() with many glVertex(), glNormal() etc. calls for every vertex in between. This keeps sending a lot of information from CPU to GPU.

There are alternatives to this. One simple alternative is to use OpenGL display lists.

Display Lists

In display lists, you render something with regular glBegin()/glEnd() calls, and tell OpenGL to remember this in the GPU. Then, whenever you want to render it again in the future, you just tell OpenGL to re-render what you had rendered and saved into a display list. You do this with the following commands:

```
GLuint list = glGenLists (1); //creates an OpenGL display list
glNewList(list, GL_COMPILE); //gets the list ready for recording
glBegin()/glEnd()...         //actual calls to render the model
glEndList();               //stops recording
```

and when you want OpenGL to re-render this list,

```
glCallList(list);
```

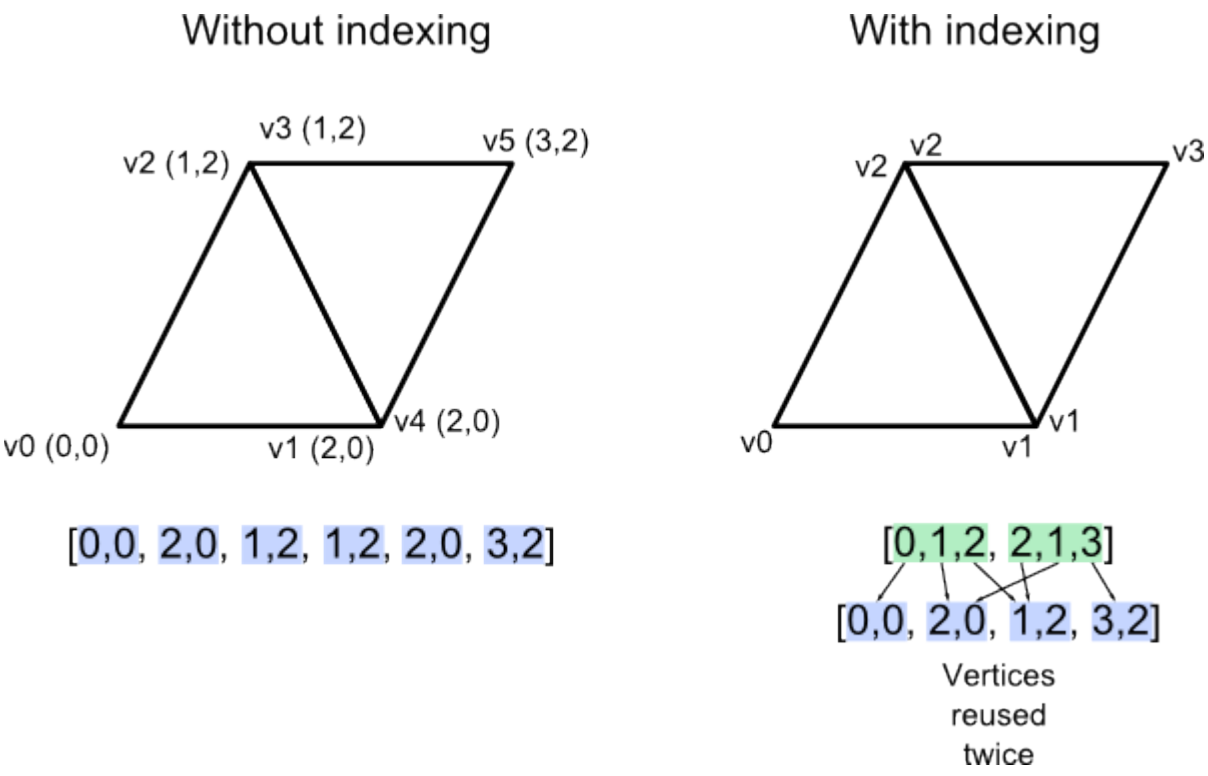
This way, you don't have to send many vertices to the GPU, they are kept there from the last time.

While display lists are useful, they can only replay a list of OpenGL commands that you cannot change. Vertex buffer objects help with that.

Vertex Buffer Objects

Instead of using OpenGL immediate mode (glBegin()/glEnd() calls), vertex buffer objects (VBOs) help you load all the data into the GPU and tell it to use them. You can later change parts of this data. Therefore, unlike display lists, it is not static.

You give a list of 3D positions, normals, texture coordinates for each vertex of your model, and then give an index array. The index array tells OpenGL to use which ones of these vertices as the faces of your model.



This not only saves you from sending many commands to the GPU, it also saves you space.

Implementing VBOs is a little more complicated than OpenGL immediate mode, though. Here is an overview of how you create a VBO:

```
// Initialization:
glGenBuffers(); // create a buffer object
glBindBuffer(); // use the buffer
glBufferData(); // allocate memory in the buffer
<stuff to get data into the buffer>
glVertexPointer(); // tell OpenGL how your data is packed inside the buffer
```

Here is how you tell OpenGL to draw using a VBO:

```
// Draw:
glBindBuffer(); // use the buffer
glEnableClientState(); // enable the parts of the data you want to draw
glDrawArrays(); // the actual draw command
```

For an actual example, please look here and here.

In OpenGL ES (and WebGL that is based on it), immediate mode does not exist anymore and you have to use VBOs instead.

Different Ways of Shading

As a part of our practical training, we learned about how OpenGL renders 3D geometry using lighting. What OpenGL implements is just a subset of shading methods and it is useful to put this into perspective.

Here are some important shading methods, from simple to complicated:

Flat Shading

In flat shading, every face has only one normal. Therefore, you cannot average normals in between faces and all the surfaces have visible corners and edges. Lighting equations are calculated once for each face.

You calculate only one color for every face.

OpenGL enables this with glShadeModel(GL_FLAT)

Gouraoud Shading

In Gouraoud shading, faces have different normals on each of their vertices. Lighting equations are calculated once for each vertex and the colors in between are linearly interpolated.

You calculate one color per vertex and interpolate them throughout the face.

OpenGL enables this with glShadeModel(GL_SMOOTH)

Phong Shading

In Phong shading, faces have different normals on each of their vertices, and throughout the face, normals are interpolated. Lighting equations are calculated for each point on a face.

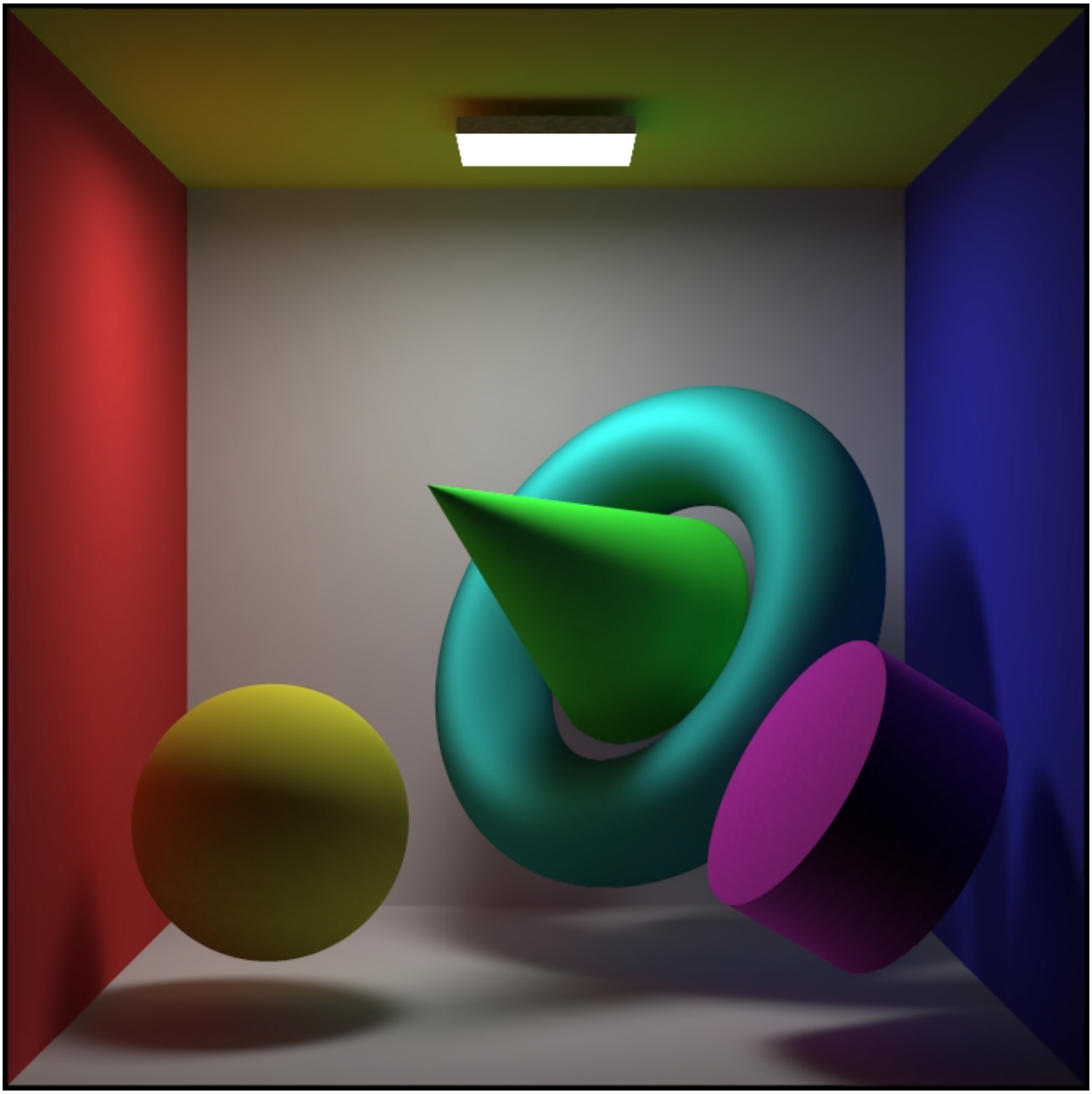
You calculate a color for each pixel that represents a face. You only interpolate the normals.

OpenGL does not support this. However, it can be implemented with custom shaders (shaders are small programs for the GPU that we did not learn about).

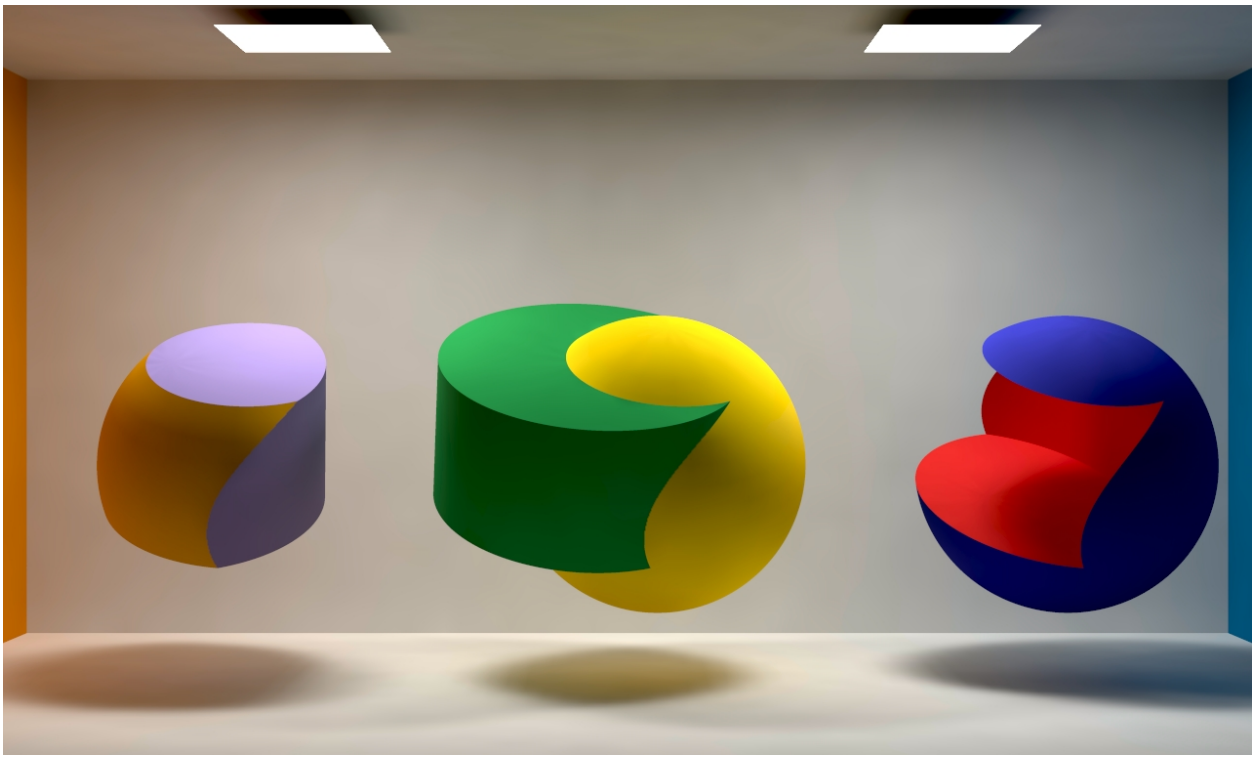
Image Souce: Tom Salter

Radiosity

Up till now, the shading models are all local illumination algorithms. That is, two objects in the scene do not effect each other's lighting unless one of them is a light source (or you have implemented shadows). Radiosity is a global illumination technique. In radiosity, the faces of polygons are separated into small regions and the relationship of each pair of such region is calculated with respect to the light. This way, we get a more realistic scene.



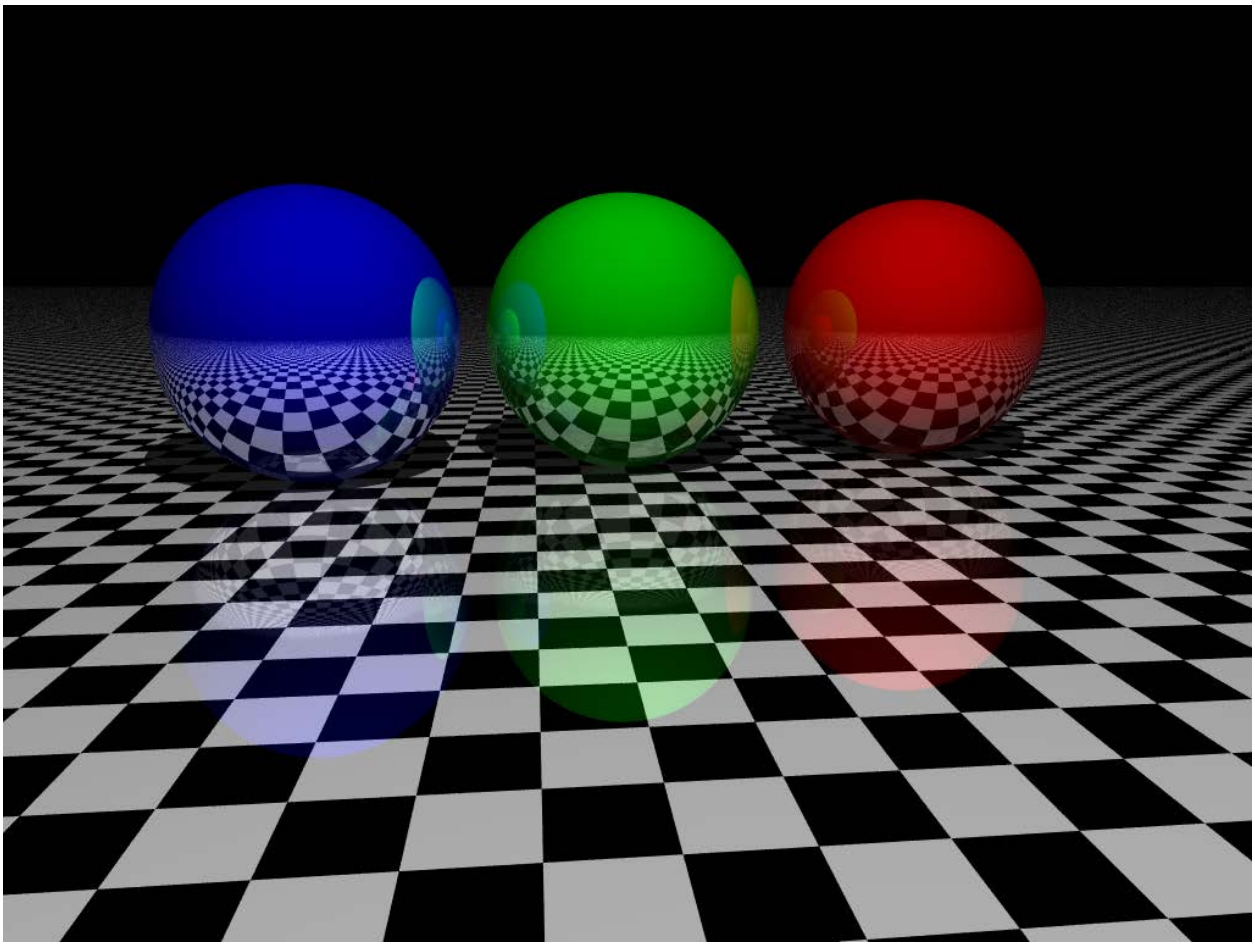
Radiosity has soft shadows and objects affecting each other's colors.



Because it requires a lot of calculations (pairing small patches on surfaces), OpenGL does not support it. However, it is possible to make radiosity work in real time with some clever tricks.

Ray Tracing

Ray tracing is even better than radiosity in that it is a simulation of light that is very close to real. Normally, light rays originate from light sources and reach our eye for us to see. Since light is reversible (calculations work the same in either direction), in ray tracing, we simulate a light ray being shot from the eye for each pixel. The light ray that bounces off different surfaces eventually reaches a light source, and the surfaces that it bounced off of determine the color of that pixel.



As you can see in the image, we can clearly see the reflection of the ground on the spheres.

As you can see, scenes rendered with ray tracing are extremely realistic, because it is an actual simulation of how light works.

Ray tracing requires a lot of computations and is generally very slow. However, there are clever tricks that make it possible to run close to real time in very advanced hardware.

Non-Photorealistic Rendering

Aside from rendering techniques that try to simulate light, there are rendering techniques that have other goals, such as mimicking cartoons, caricatures or paintings. Here are some examples:

http://classes.cec.wustl.edu/~cse452/lectures/lect14_NPR_6pp.pdf

Last Lecture

Today, we will continue the "high level gloss over" kind of lecture with the following topics. We will heavily use the internet for information. Like last week, you are responsible from this content, but in a high level. Make sure you understand the concepts and do not worry about the details.

OpenGL in the Wild

There are so many devices and platforms you can find that has OpenGL.

OpenGL ES

This is a simpler version of OpenGL that is mainly used in mobile devices. Examples: iPhone, Android, Blackberry

WebGL

This is based on OpenGL ES. You can embed OpenGL code in websites thanks to this great technology.

Alternatives to Writing OpenGL Code

We learned OpenGL because it is the most common low-level graphics library. Practically any device with real-time 3D rendering capabilities runs OpenGL. However, writing OpenGL code is not a lot of fun because it is a low-level library. When you program, you need to think in terms of high-level concepts (model) rather than low level concepts (vertices), otherwise it will be boring.

Libraries that use OpenGL

There are a lot of libraries written on top of OpenGL (libraries that call OpenGL code for you). Depending on the platform, efficiency and ease of programming requirements you have many options to choose from. Below is a nice compilation:

http://www.khronos.org/webgl/wiki/User_Contributions#Frameworks

Some notable samples are Ogre3D, OpenSceneGraph and jMonkeyEngine. For WebGL, there is three.js and philogl.

Alternative to OpenGL

The only serious alternative to OpenGL is Direct3D. It's a great choice for Windows-based systems.

Suggested Approach

"I don't know technology X" is not an excuse for a computer engineer. There are many technologies and there will be more coming up. You should use your knowledge about computer graphics and get used to quickly learning and using a new technology. Learn how to effectively use documentation, learn how to browse and learn from code. Open source is great, understand why people love it.

Shaders

Newer versions of OpenGL (and Direct3D) let you reprogram the pipeline. You should know how it works. We will follow this tutorial and use the board to explain it:

http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/pipeline33/