Computer Graphics

Discussion 3

Garett Ridge garett@cs.ucla.edu,
Sam Amin samamin@ucla.edu,
Theresa Tong theresa.r.tong@gmail.com,
Quanjie Geng szmun.gengqj@gmail.com

Part I: Some Special Matrices

Camera, Projection, and Viewport

Matrix Review

- All the objects you draw on screen are drawn one vertex at a time, by starting with the vertex's xyz coordinate and then multiplying by a matrix to get the final xy coordinate on the screen.

$$\underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{M} * \underbrace{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}_{p}$$

Transforms

$$\underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{M} * \underbrace{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}_{p}$$

- Before that matrix, the xyz coordinate is always some trivial value like (.5, .5, .5)
  - In the reference system of the shape itself
  - For example, a cube's own coordinates for its corners
- After that matrix, it's some different xy pixel coordinate denoting where that vertex will show up on the screen.
  - And z for depth, and a fourth number for translations / perspective effects
- That mapping is all that the transform does.

Transform Process

- The transform is always just one 4x4 matrix.
- But calculating what it should be involves multiplying out a big chain of intermediate special matrices. That chain is always:

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Transform Process

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Note: We never actually see the viewport matrix.
  - The viewport matrix happens out of sight at the end of the vertex shader.
  - Set up by JavaScript’s call to gl.viewport(x,y,width,height)
- All the other special matrices you do manage.

Transform Process

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The camera matrix is very much like the model transform matrix for placing shapes. But:
  - The shape being placed is the scene's observer
  - You actually use the **inverse matrix** of what you would have done to a 3D model of an actual camera

Projections

- So, there are two choices for how the view frustum is shaped: Perspective or Orthographic (parallel)
- The frustum has six planes, and the closest to the camera is called the “near plane”
- The projection matrix maps all 3D points that fall inside a frustum onto the near plane of that frustum, thereby reducing all shapes to 2D, for screen display.

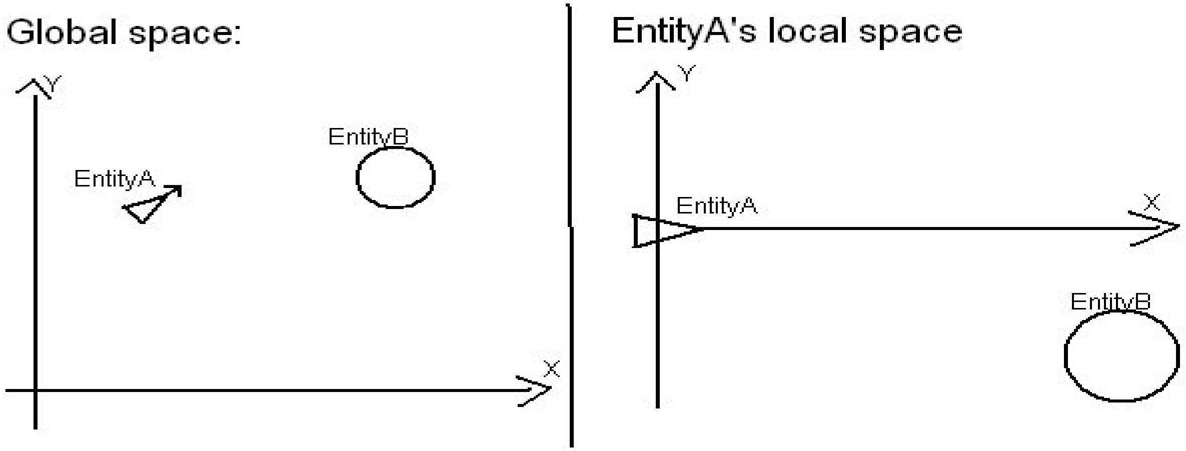
Projections: Online Demos

- http://threejs.org/examples/#webgl_camera
  - Perspective vs orthographic - the difference between the two projection frustums (and what they see) -- press O and P to switch between the two.
  - Clipping planes
  - Tons of other informative examples are linked there, like the demo of flat vs smooth shading: http://threejs.org/examples/#webgl_morphnormals

Part II: Coordinate spaces

World Coordinates

- The common coordinate system for the scene
- Also called World Space / Global Space

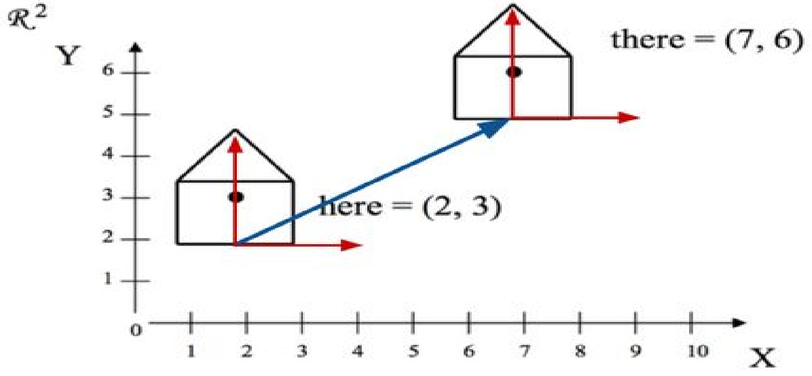


World Coordinates

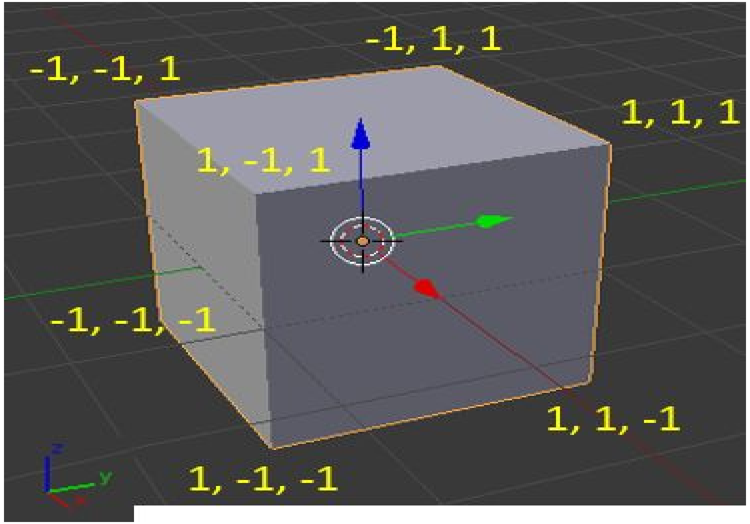
Placing object coordinates in the world

Place the coordinate system for the object in the world

- Don't know or care about the shape of the object



Object Coordinates: Transforming *every point* of an object



Object Coordinates

Object Coordinates

Each object is defined using some **convenient** coordinates

- Often “Axis-aligned”. (when there are natural axes for the object)
- Origin of coordinates is often in the middle of the object
- Origin of coordinates is often at the “base” or corner of the object
- E.g. house in previous example was (-1,0), (1, 0)…

Notice: build, manipulate object in object coordinates, also called **Object Space/Local Coordinates**

- Don’t know (or care) where the object will end up in the scene.

Screen coordinates

- So far we only have considered:
  - World Coordinates == Screen Coordinates
- We need a camera transform to draw the world from anywhere besides the world origin
  - Places a camera in the world via incremental movements (applying matrices)
  - “Bases” vs “Points” philosophies matter here
  - In “points” thinking, you’re instead “re-orienting the world” to get a particular vantage point

Remember the lecture slide about "points" thinking vs "bases" thinking:

Rule of Thumb

Transforming a point P:

Transformations: T_1, T_2, T_3

Matrix: $\mathbf{M} = \mathbf{M}_3\ \mathbf{M}_2\ \mathbf{M}_1$

Point transformed by $\mathbf{M}P$

Each transformation happens with respect to the **same** coordinate system

Transforming a coordinate system:

Transformations: T_1, T_2, T_3

Matrix: $\mathbf{M} = \mathbf{M}_1\ \mathbf{M}_2\ \mathbf{M}_3$

Transformed point has coordinates $\mathbf{M}P$ in original coordinate system

Each transformation happens with respect to **previous** coordinate system

Look at this again:

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The camera matrix is very much like the model transform matrix for placing shapes. But:
  - The shape being placed is the scene's observer
  - You actually use the **inverse matrix** of what you would have done to a 3D model of an actual camera

Camera and Model Transform

- Camera transform goes to the left of model transform since the camera is more of an **initial** basis
  - (because “bases” philosophy **starts** on the left of end of the matrix product, and starts evaluating from there)
- Unlike the model transform, a camera transform takes the thing in question (the camera) and does the matrix inverse of what you want to do to it

Matrix Inverse

- Recall this property of the matrix inverse:
  - $(M_1*M_2*M_3)^{-1} = (M_3)^{-1}* (M_2)^{-1}*(M_1)^{-1}$

When you invert a product,

– The order gets reversed

AND

– The individual matrices get inverted

The Camera Transform

- Given $C_1 * C_2 * C_3$, a sequence of transforms we'd like to do to the camera, we really get:

$$(C_1 * C_2 * C_3)^{-1}\, M_1 * M_2 * M_3[p]$$

- Which equals:

$$(C_3)^{-1} * (C_2)^{-1} * (C_1)^{-1}\, M_1 * M_2 * M_3[p]$$

- So unlike the model transform, all of our first-person camera's new incremental changes come in from the left (pre-multiplying; build the camera transform right to left)

Interpreting a written-out camera-model product

- Notice:

The first warp we apply to the image...

$$(C_3)^{-1} * (C_2)^{-1} * (C_1)^{-1} M_1 * M_2 * \mathbf{M_3}[p]$$

...is also the last (most recent) coordinate system change

Interpreting a written-out camera-model product

- Notice:

And the first warp we apply to an *image of a camera*...

$$(C_3)^{-1} * (C_2)^{-1} * (C_1)^{-1} M_1 * M_2 * M_3[p]$$

Is also the last (most recent) camera repositioning

The Camera Transform

- Conclusion:
  - When thinking of a camera as first-person (as opposed to third-person, where we actually just spin the scene around, which is like doing more to model_transform):
    - We incrementally adjust the camera over time, the increments come in as matrices from the *left*.
    - Calls to mult() will have to pre-multiply instead of post-multiply to do what we expect.

The Camera Transform

- More things to notice if you want to zoom in or out:
  - Camera transforms and model transforms both often have the form T * R *T * R * ... * T * R * **<u>S</u>**
  - So after inversion and combining them both, it looks like:

$$S^{-1}R^{-1}T^{-1}R^{-1}T^{-1}*TRTRS*[p]$$

Scales the object

Scales the view frustrum

Part III: Bee assignment

AKA frequently asked questions

Hierarchy clarifications

- Mainly meant to help you think in terms of hierarchy - the actual coding requirements are minimal
  - Post multiplication satisfies your hierarchical design constraint; methods/function calls satisfy your coding constraint
  - You should think of the bee parts themselves as being in a hierarchy (whatever part you draw first influences the matrix transformations of all parts drawn after)

Methods vs Functions

- Methods in JS work the same as they do in C++; the template uses methods
  - Use them for you bee parts as well for simplicity, since just calling this.method_name(...) will give you the correct “this”
  - With regular functions, you’d need to make sure “this” context is correct or else any shape draw call will throw and error (you’ll need to use apply or call to pass “this”)

Methods vs Functions

- From Piazza post 31 -- Things to watch out for when adding your own method:
    - The best existing code to mimic to accomplish adding your own method is actually found in the "Surfaces_Tester" class a bit farther down in the example scenes file. The key things to notice are the syntax for how it declares the class method called 'draw_all_shapes()'. Just re-use that syntax.
    - What is this "list" of methods you see? If you look at the surrounding code, what's really going on is you're passing in something to some function "Declare_Any_Class()." The second argument it wants is a list of all the desired class methods. The list is of JavaScript type "Object", like those declared with curly braces { }.
    - Pay special attention to the fact that there is a comma before the “'draw_all_shapes()” declaration (because it's actually a key-value pair in an object declaration, and those are separated by commas).
    - Also pay special attention to how the method is called. Notice the "**this**". The function "display()" of the Surfaces_Tester has to say "**this**.draw_all_shapes()" to call its own method. JavaScript is not smart enough to infer that you're trying to make the current object do something. You have to type "**this**." yourself. A lot of people will have errors about their function being undefined because they will forget to put the "**this**.".

Variables and model transform

- model_transform is just a normal local variable - nothing special about it
  - You can make your own mat4 variables and use them to store copies of model_transform
  - Allows you to quickly jump back to a previous matrix- no need for stacks (this method is actually easier than stacks for this assignment)

Recall using stacks to maintain matrix “history”:

When you do a complicated series of transformations, you want to be able to get back to where your model_transform was before. You could handle it by undoing all your transformations, or you could just make a stack:

- var stack = [];
    - Now just push your current model_transform matrix before a transformation and pop it back afterwards

stack.push(model_transform);

model_transform = stack.pop();

Better than stacks: Function calls

- You should know about what the “program stack” or “call stack” is in this course. Google if unsure.
- Every time a function gets called (in any language), a stack is being used behind the scenes.
- This stack can save you from having to use your own.
- Every time a function ends, it’s very similar to backing up one level in the history of your model_transform variable, which resumes its value in the caller’s scope

Better than stacks: Function calls

- Conclusion: Instead of a stack operation, use a function instead. Write your own function and pass in model_transform by value, then let that function handle its own copy; when you're done with that branch of your shape, back up one level simply by letting the function finish.
- In our template, you would add class methods.
- The emphasis on stacks in graphics courses is just a vestigial holdover from how old (pre-shader) graphics cards used to work.
    - They managed their own stacks of matrix history, and you had to go through it instead of controlling it yourself. Very slow back-and-forth between CPU and GPU. It was inflexible. Modern GPUs run programs that <u>you</u> write.