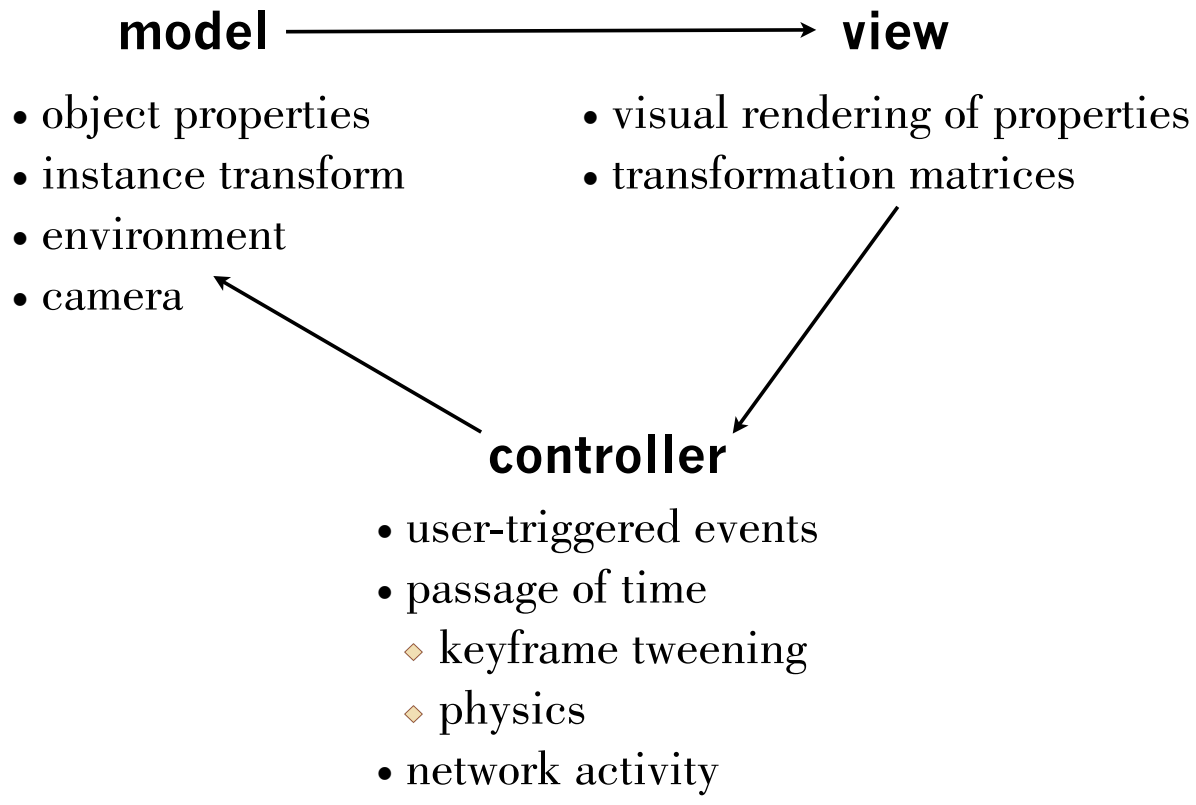
Interaction & Dynamic Behavior Strategies

- So now you have a 3D scene, populated with interesting shapes—what next?
- What's next is interaction and/or dynamic behavior: making your scene change in response to user activity and/or physics and/or the passage of time
- This may seem complicated or intimidating, but ultimately everything falls under a pattern that should look familiar:



Controller Strategies

- We start with discussing the controller because this is the entry point, code-wise, of any interactive or dynamic behavior
- The sample code that you have seen already provides the general "scaffolding" for two of the major controller "triggers:" user events and the passage of time
- Structurally, network activity can fit either model—if you poll for activity, that's passage-of-time style; if you implement push, it's like handling user events

User-Triggered Events

- For WebGL, this is the same as for any web application —bind a function to an element for an event
- The event listener's modifies your application's model based on the triggered event—e.g., an object's instance transform, the camera, new/removed objects
- After a change, re-render the scene (if not automatic)
- Mouse or touch events should be bound to the `canvas` element that holds the 3D scene; you will need to unproject their 2D coordinates to your 3D space

The Passage of Time

- Passage-of-time behavior falls into two general categories: keyframe-based tweening or movement based on a physics model
- The overall structure is the same for both—you need to invoke a scene update function at regular intervals (possibly the same function that draws the scene, but not necessarily)
- The scene update function computes the state of things for the next frame and changes the model accordingly

3D Keyframe Tweening

- What you've seen before: You need a data structure for declaring your keyframes and the object state for each frame and easing functions for calculating the in-between states of your objects
- What you need to add: Use the tweened values to change the instance transforms of your objects
  - Tweening can also be done to your camera parameters for automated "flyovers"
  - …or anything else really: colors, lights, etc.

Physics in Your Scene

- First off, bone up on Newtonian mechanics or identify a physics engine that you can use in your application
- Either way, you start by defining any relevant physics properties for your objects—velocity, acceleration, mass, forces at play
- If you are rolling your own engine, note that many calculations use vectors heavily
- Physics can be affected by user events—e.g., acceleration or braking; steering

Implementing physics is not that different from tweening:

- When your scene updater function is invoked, use your physics-related properties to compute the new states of your objects
- As before, these states can include instance transforms, the camera, or anything else
- One wrinkle is binding to the passage of real time—i.e., track the system clock so that you relocate or accelerate based on the amount of real time that has passed, and not the frame count ($S = Vt$, remember?)
- Note that this is all applied calculus and/or differential equations—that's why you took those classes :)

Model Strategies

- The controller discussion has already mentioned the model approaches you are likely to use:
  - Change instance transforms
  - Change camera values (especially location and orientation; maybe the up vector for flight sims)
  - Change the environment (lighting, viewing volume)
- What's important is that your model doesn't care how it changed—it simply tracks the needed values

View Strategies

- Similarly, the controller discussion also covered what your view code does:
  - The view walks through your scene data structure and displays what it finds there
  - For 3D scenes, many model changes result in matrix changes, so connect the dots appropriately
- Like with the model, your view code shouldn't care how the scene it is rendering got changed—it simply takes what's there and sends it to WebGL