Stage 6: video 3

Scene 1: On-Set

MVC is a good choice for structuring software applications in most cases, however is is not the only design pattern and there are times when it is not the best fit.

<MOTION>
PONG GAME

As a counter example, we'll look at the game of Pong. Applying MVC we could divide the application into a model: a simple physics engine storing position, etc… for a couple moveable boundaries (the paddles) and a moving projectile (the ball).

The view would contain the visual representation of the game: a couple crude lines (the paddles) and a small round circle (the ball), along with the paddle control interface.

As we begin implementing this design, we start to notice some troubling redundancies: for instance: the physics model of the of the ball requires: a size, a position in 2-d space, a velocity, etc... In the view, the visible ball object also has a size, a screen-position (a position in 2d space), etc..

In the case of Pong, any change in the physics model, for example the ball bouncing off a boundary, should immediately be reflected in the view. Likewise, any rotation of the paddle control should cause an immediate change in both the position of the visual representation of the paddle on the screen and the position of a boundary within the physics model. Any discrepancies will produce and nearly unplayable game. Insisting on a separation between model and view this case leads to dangerous redundancy.

Redundancy in software is dangerous. Redundancy increases complexity, reduces efficiency and risks introducing bugs associated with data inconsistency. Like normalization in databases, information should be stored once if possible.

Looking at the problem, it becomes clear that the application we're trying to develop would be well served by a *strong* coupling between the physics model of the game(the physics engine) and what needs to be displayed on the screen. Any change in the physical state of an object (like the ball) should immediately be displayed in the interface. Likewise, interface interaction (moving of paddles) should directly affect the model (constrained by the laws of the physics engine).

In this case, forcing a decoupling between the model and view is actually counter-productive.

</MOTION>

For an example in Cocoa, we'll take a brief look at sprite creation in Apple’s Sprite Kit framework.

Scene 2: On-Set

<MOTION>
Sprite graphics

Much of historical game development has involved two-dimensional images or animations representing game objects. Mario (of Donkey-Kong fame), Pac-Man and the spaceships in Galaxian are all good examples of sprites.

</MOTION>

Sprite Kit is a framework that was introduced in iOS 7 designed to simplify sprite programming for games and other applications.

As with the objects in our pong example, when programming sprites, there *should be* a tight coupling between the *visual* representation of the sprite and the sprite’s state within the model underlying the game mechanics. For example when displaying a “Mario” type character, his velocity and acceleration in the physics engine are *intimately* tied to the movement of his visual representation across the screen as well as associated character animations.

<KEYNOTE>
Here is a short code snippet for creating a sprite using the SpriteKit framework:

```
SKSpriteNode *player;
player = [SKSpriteNode spriteNodeWithImageNamed:@”shortItalianPainterGuy”];
player.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize: guy.size ];
player.physicsBody.mass = 100.0;
```

SKSpriteNode is the sprite to be created (they’re called “nodes” because sprites exist within a scene hierarchy). ‘Player’ is a diminutive character in overalls, so it’s initialized using an image “shortItalianPainterGuy”.

The physics of the sprite are set using the physicsBody property, and we’ve specified the size and mass.

The Sprite Kit framework clearly assumes a strong coupling between a sprite's appearance and its physics attributes.

</KEYNOTE>

Other variations on Model-View-Controller include: Model-View-ViewModel, Model-View-Adapter and Model-View-Presenter. If you're curious about these there are resources available in the Teacher's Notes.

Hopefully, through watching these videos, you've come to know and appreciate design patterns a bit more. Thanks for watching.

Questions:

1. True or False. All iOS applications should conform to MVC. (ans. false)
2. True or False. Redundancy is a good thing in software. (ans. false)
3. Which version of iOS introduced Sprite Kit? (ans. iOS 7)
4. The SKSpriteNode property used to modify a sprite's attributes within the physics engine is? (ans. physicsBody)
5. True or False. MVC is the proper choice of design pattern if you're implementing pong. (ans. true)