CSc 165 Lecture Note Slides

Camera Control In Games

Dr. John Clevenger
Computer Science Dept.
CSUS

Overview

- 6 DoF vs. Constrained Cameras
- "MouseLook" Mode and Cursors
- 1st-person vs. 3rd-person Cameras
- Chase Cameras
- Heads-Up Displays (HUDs)
- Viewports

Unconstrained (6 DoF) Cameras

- Consider the following camera sequence:

```
Rotate(90,Y)
Rotate(90,X)
Rotate(-90,Y)
Rotate(-90,X)
```

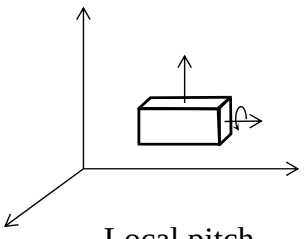
- Does it put the camera back to initial state?
  - Why not?
- The same effect creeps into camera control in small (but *cummulative)* amounts with small rotations…

Run

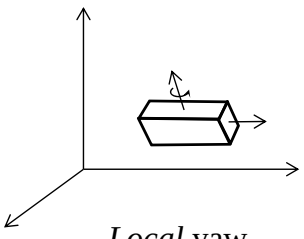
John Clevenger
CSc Dept, CSUS

Constrained Cameras

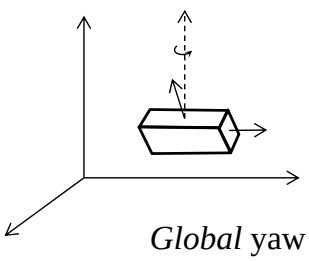
- 6 DoF "flight": pitch + yaw introduces *roll*
  - Exactly correct for "flight simulators" or "spaceships"
  - May not be appropriate for ground-based FPS games (*looking around* shouldn't cause *roll*)
- Can be controlled by using "local pitch" but "***global*** yaw"



Local pitch



Local yaw



Global yaw

Run

John Clevenger
CSc Dept, CSUS

"Mouse-Look" Mode*

- "Mouse-look" == using mouse to control camera orientation (introduced in "Quake" c. 1996)
- Two methods of obtaining mouse moves:
  - Input Manager *axis* devices (direct)
  - Window Manager mouse listener routines (indirect)
    - AWT MouseMoved, MouseDragged, etc.
- Problem with WM mouse control:
  - Mouse stops at screen edge
  - Player can't move camera beyond that limit
  - Solution: *recenter* mouse after each move

*Also known as "Free-Look" mode

The Robot Class

```
/** This class demonstrates how to use a "Robot" to recenter the mouse after every
 *  mouse move, keeping the mouse from ever reaching the screen edge.  The class is
 *  based on the Sage game engine, but the Robot can be used in other contexts as well.*/
public class RobotMouseLook extends BaseGame
                            implements MouseListener, MouseMotionListener  {
    private Robot robot;
    private Point canvasCenter;              //center of the rendering canvas
    private float prevMouseX, prevMouseY;    //location of the mouse prior to a move
    private boolean isRecentering;           //indicates the Robot is in action
    ...
    protected void initGame() {
        ...
        initMouseMode();
        ...
    }
    /** This method creates a Robot to control mouse recentering, and invokes a
     *  local method to use the Robot to center the mouse initially.  */
    private void initMouseMode() {
        Dimension dim = getRenderer().getCanvas().getSize();
        canvasCenter = new Point(dim.width/2, dim.height/2);
        isRecentering = false;
        try {
            robot = new Robot();
        } catch (AWTException ex) {   //some platforms may not support the Robot class
            throw new RuntimeException("Couldn't create Robot!");
        }

        recenterMouse();
        prevMouseX = canvasCenter.x;  //'prevMouse' defines the initial
        prevMouseY = canvasCenter.y;  //   mouse position
    }
    //continued
```

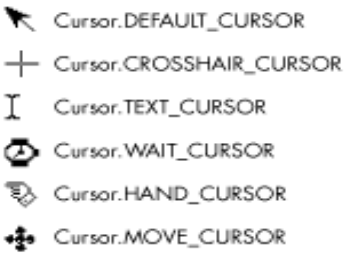
The Robot Class (cont.)

```
    //class RobotMouseLook continued
    /** This method uses the Robot class to position the mouse at screen center */
    private void recenterMouse() {
        //convert the canvas-relative center point to screen coordinates
        Point p = new Point(canvasCenter.x, canvasCenter.y);
        Canvas canvas = getRenderer().getCanvas();
        SwingUtilities.convertPointToScreen(p, canvas);
        //show that we're about to ask the robot to recenter the mouse
        isRecentering = true;
        //use the robot to move the mouse.  Note that this will generate (one)
        // MouseEvent whose absolute location coordinates will be p – the center point.
        robot.mouseMove(p.x, p.y);
    }
    /**This method uses mouse movement to alter the camera, unless it was a robot move*/
    public void mouseMoved(MouseEvent e) {
        // check whether the robot is recentering; if so and whether the MouseEvent
        //   location is the center; if so, this event is from the robot
        if (isRecentering && canvasCenter.x == e.getX() && canvasCenter.y == e.getY() ) {
            // yes, mouse has been recentered; show that recentering is complete
            isRecentering = false;
        } else {   // not recentering; event was due to a user mouse-move; process it
            curMouseX = e.getX();
            float mouseDeltaX = prevMouseX - curMouseX;
            yaw(mouseDeltaX);      //local routine to yaw the camera
            prevMouseX = curMouseX;
            curMouseY = e.getY();
            float mouseDeltaY = prevMouseY - curMouseY;
            pitch(mouseDeltaY);    //local routine to pitch the camera
            prevMouseY = curMouseY;
            // tell robot to put the cursor back to the center (since user just moved it)
            recenterMouse();
            prevMouseX = canvasCenter.x;  //reset 'prev' to center
            prevMouseY = canvasCenter.y;
        }
    }
    //other methods here (pitch, yaw, mouseListeners, ...)
}
```

Run

Setting Mouse Cursors

- **Some Java pre-defined cursors:**



- **Obtaining a cursor:**

```
Cursor waitCursor =
  Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR);
```

- **Changing the current cursor :**

```
renderer.getCanvas().setCursor(waitCursor);
```

Setting Mouse Cursors (cont.)

- **Defining your own custom cursor**

```
Cursor cursor = Toolkit.getDefaultToolkit().
        createCustomCursor(Image i, Point hotSpot,
                           String name);
```

Example:

```
Image pencilImage =
        new ImageIcon("images/pencil.gif").getImage();
Cursor pencilCursor =
        Toolkit.getDefaultToolkit().
              createCustomCursor(pencilImage,
                                 new Point(0,0),
                                 “PencilCursor"  );
```


Setting Mouse Cursors (cont.)

- **Invisible cursors**
  - Not predefined in Java
  - Can be created using an “undefined image”

```
Toolkit tk = Toolkit.getDefaultToolkit();
Cursor invisibleCursor =
   tk.createCustomCursor(tk.getImage(""),
                         new Point(),
                          "InvisibleCursor");

renderer.getCanvas().setCursor(invisibleCursor);
```

Run

1P vs. 3P Cameras

- First-Person (1P) Cameras
  - Located at the player's "point of view"
  - Manipulating *camera* changes *player's* loc/view-dir
- Gaming characteristics of 1P:
  - Good for "local environment" feedback sounds
    - Heartbeat, breathing, footsteps, weapon sounds
  - Provides limited view of surroundings
    - Things can "sneak up" (good for building *suspense*)
  - Easier to "aim" in shooting games

Types of 3P Cameras

- Bird's-eye ("2 ½ D" perspective)
  - Fixed camera looking down on a (mostly) 2D world
  - Player avatar (*if any*) is independent of camera

Examples:

Sim City 2000

Starcraft

League of Legends

Types of 3P Cameras (cont.)

- Chase (also called *tracking)*
  - Camera follows avatar, maintains constant relative view ("over-the-shoulder"; "behind-the-back")
  - Camera typically on "springs" to reduce jerkiness

Examples:

Mario Kart

Mario Kart 64 Battle Mode

Types of 3P Cameras (cont.)

- "Targeted" (also called *orbit*)
  - Camera always looks at avatar, but can be independently controlled in various ways (orbit, zoom) and may also affect avatar

Example: World of Warcraft

Camera *behind* avatar

Camera orbited *around* avatar

Building a Targeted 3P Camera

- Camera characteristics:
  - Location: typically starts "above" and "behind" avatar
  - Focal point: usually directed *at (or slightly above)* avatar

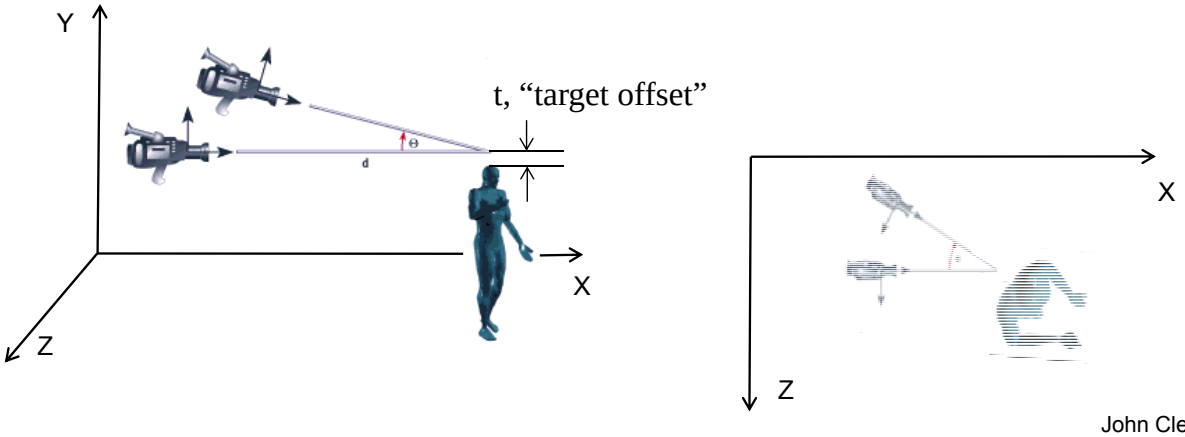


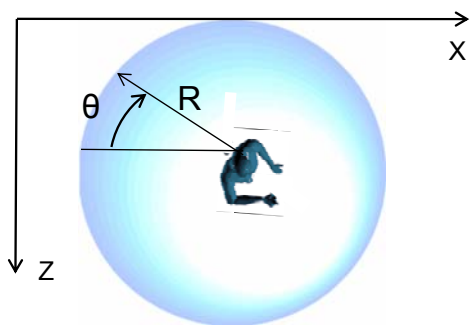
Image credit: www.gamasutra.com

John Clevenger
CSc Dept, CSUS

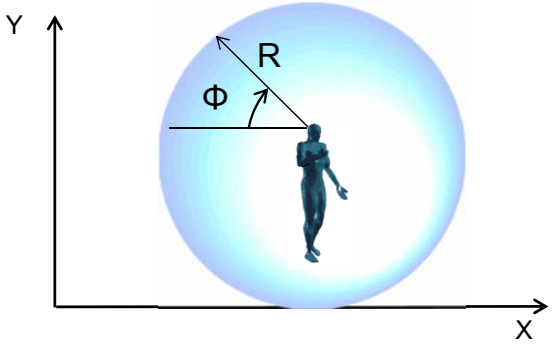
3P Camera Positioning

- Orbit camera position defined in *spherical* coordinates:

Azimuth **θ**, altitude (elevation) **Φ**, radius (distance) **R**



Azimuth

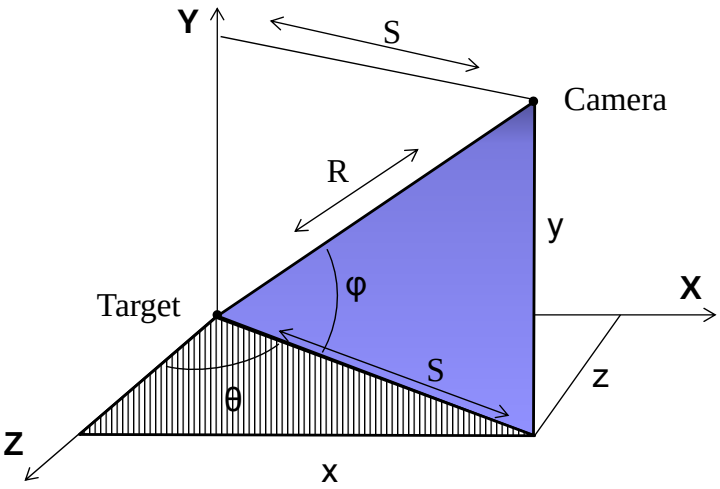


Elevation

Image credit: www.gamasutra.com

John Clevenger
CSc Dept, CSUS

Computing Spherical Position



$$S = \sqrt{x^2 + z^2} = \mathrm{R}\cos(\varphi)$$

$$R = \sqrt{S^2 + y^2} = \sqrt{x^2 + y^2 + z^2}$$

$$x = S\sin(\theta) = R\cos(\varphi)\sin(\theta)$$

$$y = R\sin(\varphi)$$

$$z = S\cos(\theta) = R\cos(\varphi)\cos(\theta)$$

John Clevenger
CSc Dept, CSUS

Targeted Camera Avatar Control

- Typical controls:
  - ASWD moves ***avatar*** (3P camera "follows")
  - Mouse X/Y controls camera *azimuth* and *elevation*
  - Mouse *wheel* controls *distance* (zoom)
  - Avatar may *rotate* with camera rotation (e.g. when right mouse button is down)

Run

John Clevenger
CSc Dept, CSUS

3P Camera Controller

```
/**This class  defines an object which manages an "orbit camera"; that is, a third person
 * camera which always looks at a given target and which can be "orbited" around
 * the target using the mouse or other controller. */
public class ThirdPersonCameraController {

    private ICamera cam;                          //the camera being controlled
    private SceneNode target;                     //the target the camera looks at
    private float cameraAzimuth;                  //rotation of camera around target Y axis
    private float cameraElevation;                //elevation of camera above target
    private float cameraDistanceFromTarget;       //distance between camera and target
    private Point3D targetPos;                    //target's position in the world
    private float targetRotation;                 //rotation caused by user mouse input
    private Matrix3D initialTargetRotation;       //rotation value initially defined in target

    public ThirdPersonCameraController(ICamera cam, SceneNode target,
                                       IInputManager inputMgr, String controllerName) {
        this.cam = cam;
        this.target = target;
        initialTargetRotation = target.getLocalRotation();
        cameraAzimuth = 180;            //start from BEHIND the target
        targetRotation = 0;             //target rotation accumulated by this class

        updateTarget();                 //init our notion of target values
        updateCameraPosition();         //set initial camera position behind target
        cam.lookAt(targetPos, worldUpVec);
        setupInput(inputMgr, controllerName);  //assigns Actions so controller movements
                                               //increment azimuth, elevation and distance
    }
    //continued...
```


3P Camera Controller (cont.)

```
    /** Updates this ThirdPersonCameraController by determining the current target position,
     * setting a new camera position based on the target position and the current azimuth,
     * elevation, and distance for the camera (relative to the target), and forcing the
     * camera to look at the target.  Intended to be called by gameLoop update() */
    public void update(float time) {
        updateTarget();                 // update the target rotation and position information
        updateCameraPosition();         //move camera to proper azimuth, altitude, and dist
        cam.lookAt(targetPos, worldUpVec); // look at target from the current camera position
    }

    /** Updates the local information about the target position based on the target's current
     * world translation, and updates the target's rotation based on the rotation information
     * accumulated by this class. */
    private void updateTarget() {
        //update our local information about where the target is located
        targetPos = new Point3D(target.getWorldTranslation().getCol(3));
        targetPos = new Point3D(new Vector3D(targetPos).add(targetOffset));

        //reset the target rotation to its initial value
        target.setLocalRotation((Matrix3D)initialTargetRotation.clone());
        //add the current user-specified rotation to the target
        target.rotate(targetRotation, worldUpVec);
    }

    //continued...
```

3P Camera Controller (cont.)

```
/** Updates the camera position by computing its azimuth, elevation, and distance
 * relative to the target in spherical coordinates, then converting those spherical
 * coords to Cartesian coordinates and setting the camera position from that.
 */
private void updateCameraPosition() {
    //get the total rotation to put the camera at the desired azimuth around target
    double theta = cameraAzimuth;

    // get the altitude angle phi
    double phi = cameraElevation ;

    // get the distance r between the camera and target
    double r = cameraDistanceFromTarget;

    // the set (theta, phi, r) is now the spherical coord of the desired
    // camera position; convert it to world coords
    Point3D desiredCameraLoc = convertSphericalToCartesian(theta, phi, r);

    // move the camera to the "desired location"
    cam.setLocation(desiredCameraLoc);

}

//continued ...
```

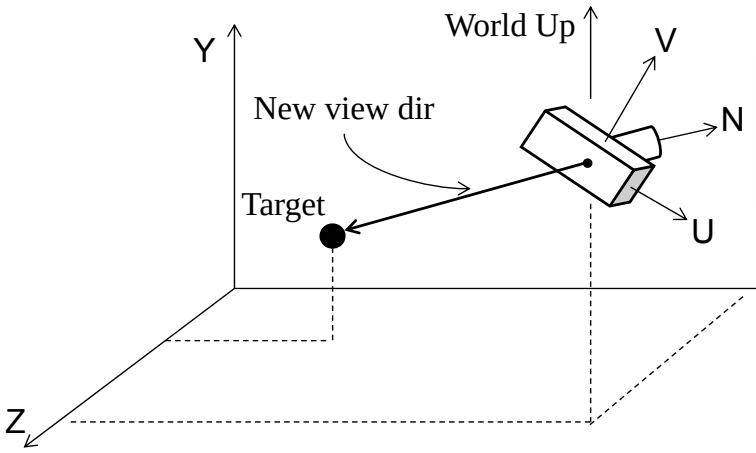

3P Camera Controller (cont.)

```
/** This class moves the camera around the target (changes camera azimuth). An instance
 *  of this class is added as an Action on the Mouse X axis.  */
private class OrbitAroundAction extends AbstractInputAction {

    public void performAction(float time, Event evt) {
        //only respond to mouse actions if a button is pressed
        if (!leftPressed && !rightPressed) {
            return;
        }
        //determine the amount of rotation
        float eventValue = evt.getValue();
        float rotAmount = eventValue * mouseSpeed;
        //update camera azimuth from mouse input
        cameraAzimuth += rotAmount ;
        cameraAzimuth = cameraAzimuth % 360 ;
        //if right-button is pressed, also rotate the target by the same amount
        if (rightPressed) {
            // update the target rotation
            targetRotation += rotAmount;
            targetRotation = targetRotation % 360 ;
        }
    }
}
//other private classes and methods here to support addition mouse controls
} //end ThirdPersonCameraController class
```

Computing *Look-At*

- Given: a camera *position* and *orientation* (U,V,N axis directions)
- Needed: function **`lookAt(target,worldUp)`**



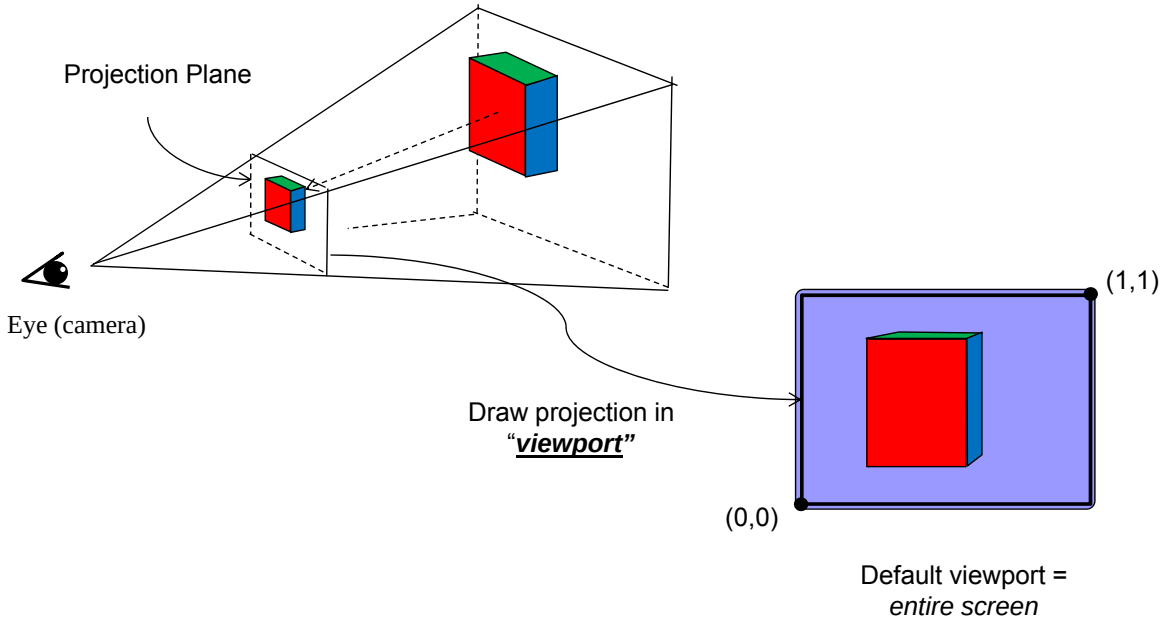
$$viewDir = tPos - cPos$$
$$rightDir = viewDir \times worldUp$$
$$upDir = rightDir \times viewDir$$

John Clevenger
CSc Dept, CSUS

Multi-Player "Split-screen"

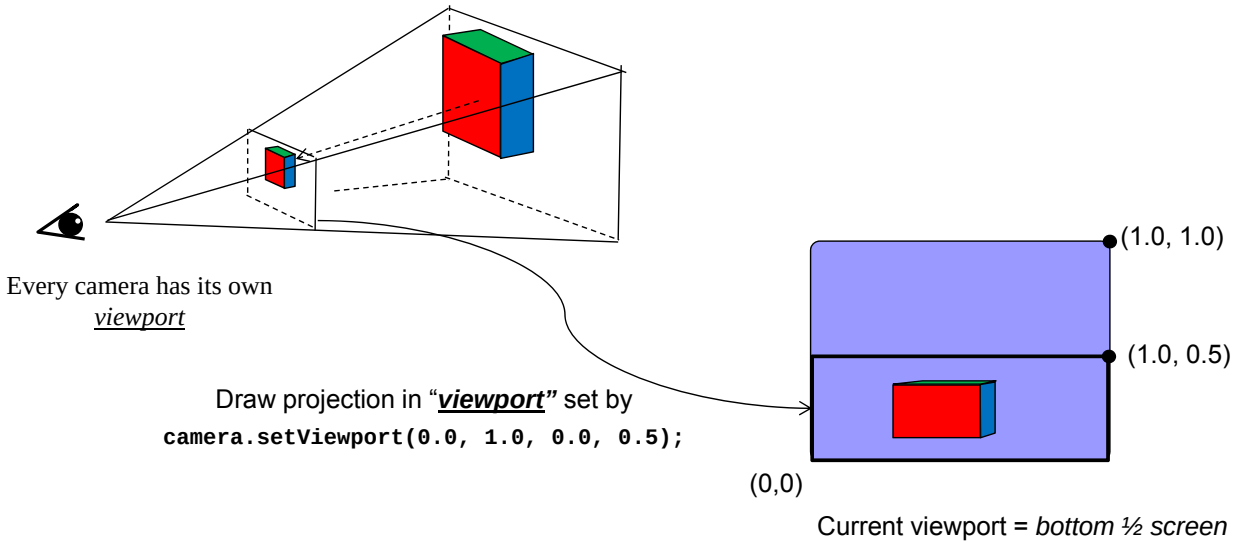
John Clevenger
CSc Dept, CSUS

Viewports



Changing the Viewport

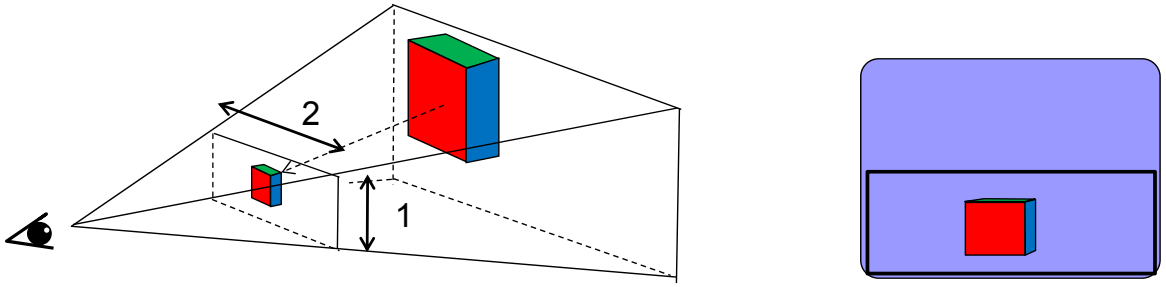
```
camera.setViewport (left, right, bottom, top)
```



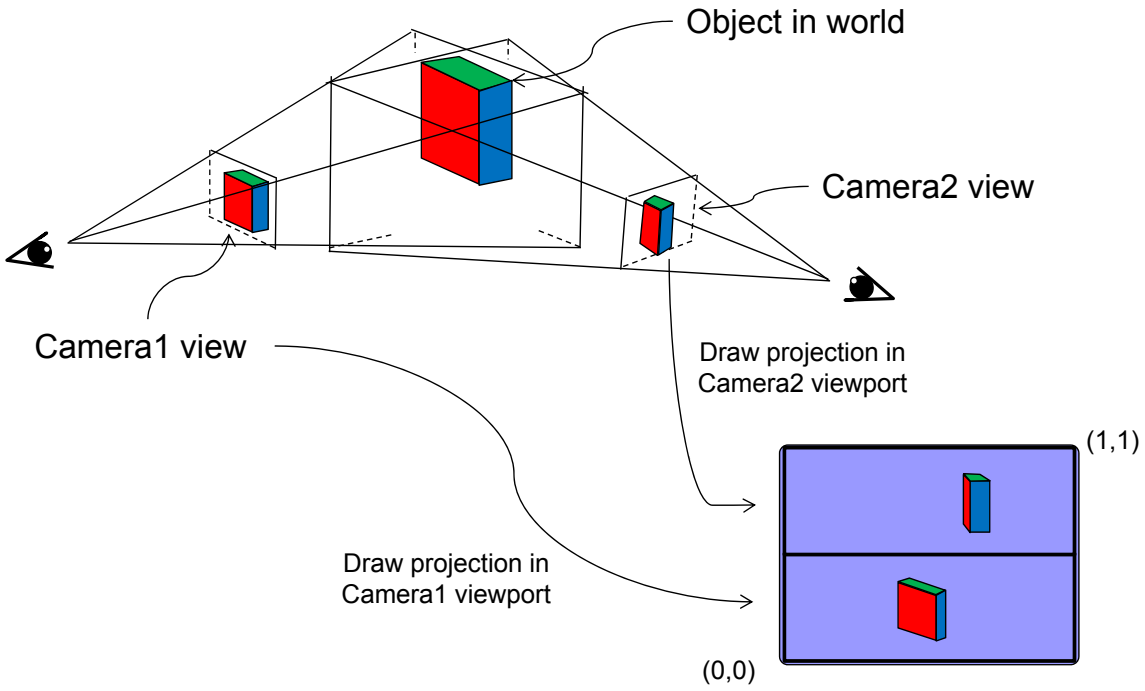
Avoiding Viewport Distortion

- Change the *view frustum aspect ratio* to match the *viewport aspect ratio*

```
//map the frustum to a 50%-high viewport
camera.setPerspectiveFrustum(fovY, 2.0, near, far);
camera.setViewport(0, 1, 0, 0.5);
```



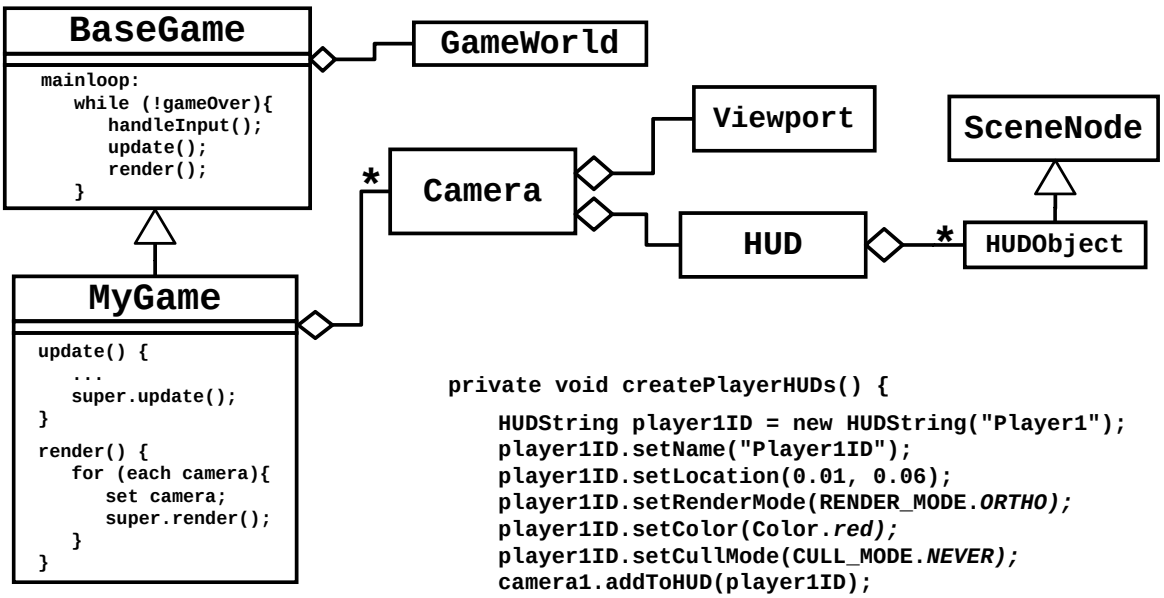
Multiple Cameras



Multiple Cameras (cont.)

- Game maintains a *collection* of cameras, each with:
  - a *viewport*
  - a *HUD* (collection of HUDObjects)
- ***initGame()***
  - Creates cameras (one per player)
  - Associates input controls with different cameras
  - Defines HUD items for each camera
- ***render()*** draws *each camera's* view, in that camera's viewport, including camera's HUD

Camera HUDs




```
private void createPlayerHUDs() {
    HUDString player1ID = new HUDString("Player1");
    player1ID.setName("Player1ID");
    player1ID.setLocation(0.01, 0.06);
    player1ID.setRenderMode(RENDER_MODE.ORTHO);
    player1ID.setColor(Color.red);
    player1ID.setCullMode(CULL_MODE.NEVER);
    camera1.addToHUD(player1ID);
    //code here defining additional HUDObjects
}   //  for each camera...
```

Multiple Camera Demo

```
/** Demonstrates use of multiple viewports by defining two players, one whose avatar is
 * controlled by WASD and the mouse, the other whose avatar is controlled by the arrow
 * keys and a controller joystick. */
public class MultipleViewports extends BaseGame {
    private IRenderer renderer;
    private IInputManager inputMgr;
    private SceneNode player1, player2;
    private ICamera camera1, camera2;
    private ThirdPersonCameraController cam1Controller, cam2Controller;
    protected void initGame() {
        createScene();
        createPlayers();
        initInput();
    }
    private void createPlayers() {
        player1 = new Pyramid("Pyramid1");        // construct a pyramid as the avatar
        player1.translate(0, 1, 50);              //  ("target") for player1
        player1.rotate(180, new Vector3D(0, 1, 0));
        addGameWorldObject(player1);              // add player1 avatar to the game world
        camera1 = new JOGLCamera(renderer);       //define the first player's camera
        camera1.setPerspectiveFrustum(60, 2, 1, 1000);
        camera1.setViewport(0.0, 1.0, 0.0, 0.45);

        player2 = new Pyramid("Pyramid2");        // construct player2 avatar similarly
        player2.translate(50, 1, 0);
        player2.rotate(-90, new Vector3D(0, 1, 0));
        addGameWorldObject(player2);
        camera2 = new JOGLCamera(renderer);
        camera2.setPerspectiveFrustum(60, 2, 1, 1000);
        camera2.setViewport(0.0, 1.0, 0.55, 1.0);
    }
```

John Clevenger
CSc Dept, CSUS

Multiple Camera Demo (cont.)

```
//...continued
private void initInput() {
    String keyboardName = inputMgr.getKeyboardName(); //get controller handles
    String mouseName = inputMgr.getMouseName();
    String gpName = inputMgr.getFirstGamepadName();

    //wrap cameras in a 3P controller
    cam1Controller = new ThirdPersonCameraController(camera1,player1,inputMgr,mouseName);
    cam2Controller = new ThirdPersonCameraController(camera2, player2, inputMgr, gpName);

    //code here to assign Actions to controller components(e.g. keys) as needed...
}

//override BaseGame's update() to do our game-specific updates
protected void update(float time) {
    cam1Controller.update(time);
    cam2Controller.update(time);
    ...
    super.update(time);
}

//override BaseGame's render() method to allow rendering each camera view
protected void render() {

    renderer.setCamera(camera1);
    super.render();

    renderer.setCamera(camera2);
    super.render();
}
}//end class MultipleViewports
```

Run

John Clevenger
CSc Dept, CSUS