CSc 165
Computer Game Architecture

04 - Camera Control

Overview

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

"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
- Window Manager mouse listener routines
  - AWT MouseMoved, MouseDragged, etc.

* also known as "Free-Look" mode

"Mouse-Look" (continued)

Challenges with WindowManager mouse control:

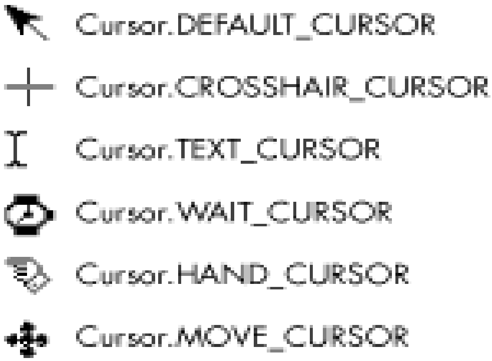
- Mouse stops at screen edge
- Player can't move camera beyond that limit

Java solution:

- *recenter* mouse after each move
- Java Robot class can be used for this
- Can also consider altering (or removing) the cursor

Setting Mouse Cursors

Some Java pre-defined cursors:



Cursor.DEFAULT_CURSOR

Cursor.CROSSHAIR_CURSOR

Cursor.TEXT_CURSOR

Cursor.WAIT_CURSOR

Cursor.HAND_CURSOR

Cursor.MOVE_CURSOR

Obtaining a cursor:

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

Changing the current cursor in TAGE:

```
rendersystem.getGLCanvas().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");
```

(note: “Toolkit” is part of Java AWT)

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");
rendersystem.getGLCanvas().setCursor(invisibleCursor);
```

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?

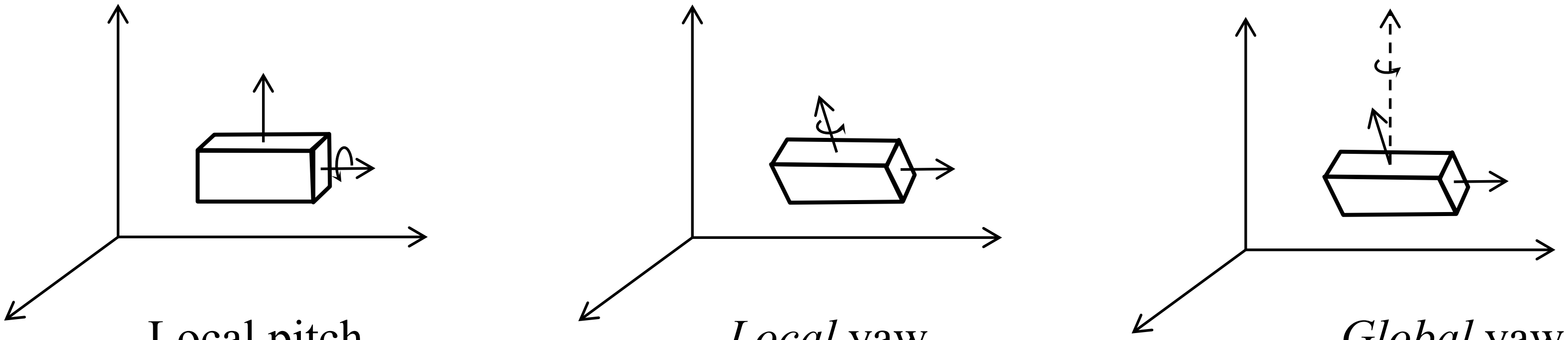
The same effect creeps into camera control in small (but *cumulative)* amounts with small rotations…

Constrained Cameras

6 DoF “flight”: pitch + yaw introduces *roll*

- Appropriate for “flight simulators” or “spaceships”
- Not appropriate for ground-based FPS games (*looking around* shouldn’t cause *roll*)

Control by using local pitch, but ***global*** *yaw*



Local pitch

Local yaw

Global yaw

Transformation matrix for *Local Yaw:*

```
Matrix4f.rotation(rotationAngleAmt, camera.getV());
```

Transformation matrix for *Global Yaw:*

```
Matrix4f.rotateY(rotationAngleAmt);
```

1P vs. 3P Cameras

First-Person (1P) Cameras:

- Located at the player's "point of view"
- Player's loc/dir changed by manipulating *camera*

Gaming characteristics of 1P:

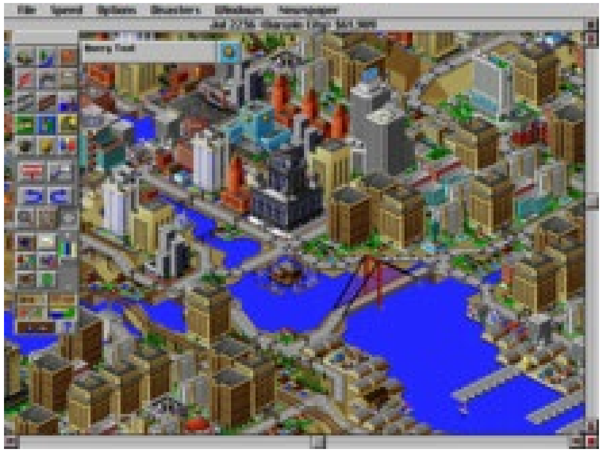
- Good for "local environment" feedback sounds
  - Heartbeat, breathing, footsteps, weapon sounds
- Provides limited view of surroundings
  - Things can "sneak up" (good for building *suspense*)
- Easier to "aim" in shooting games

Types of 3P Cameras

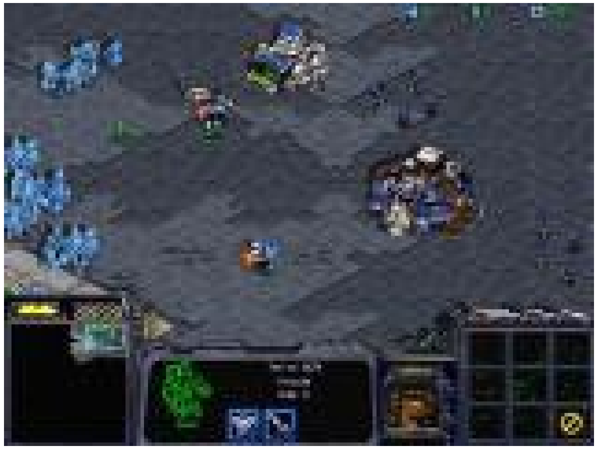
Bird’s-eye (“2 ½ D” perspective)

- Fixed camera looking down on a (mostly) 2D world
- Player avatar is independent of camera
- Some games have no avatar (e.g., building games such as *SimCity*, or Real-Time Strategy games such as *Age of Empires*)

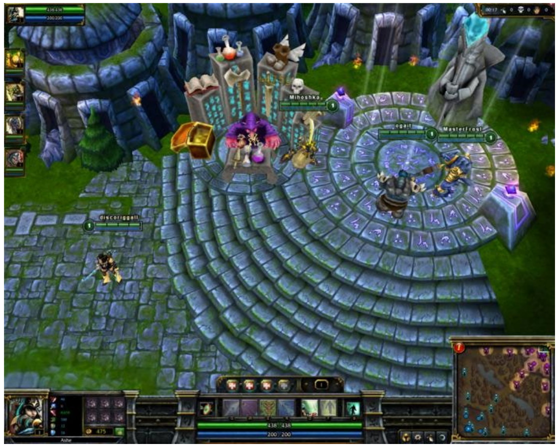
Examples:



Sim City 2000



Starcraft



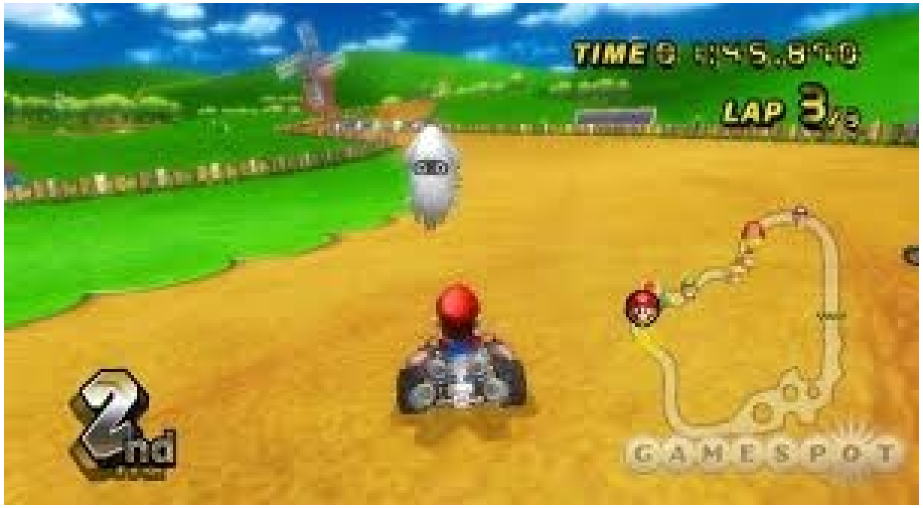
League of Legends

Types of 3P Cameras (cont.)

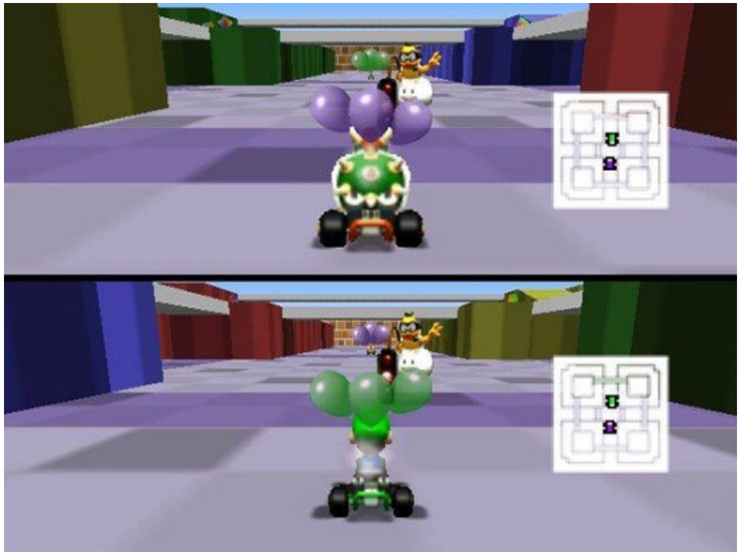
Chase (also called *tracking)*

- Camera follows avatar, maintains constant relative view (“over-the-shoulder”; “behind-the-back”)
- Camera typically on “springs” to reduce jerkiness

Examples:



Mario Kart



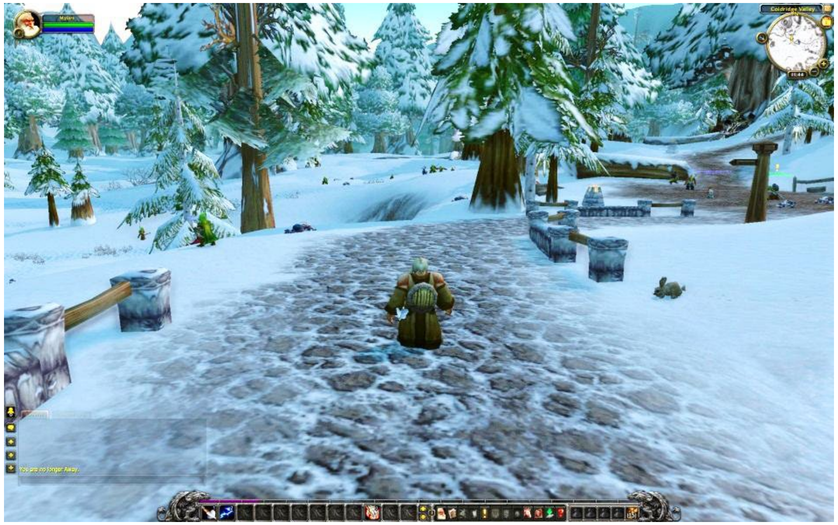
Mario Kart 64 Battle Mode

Types of 3P Cameras (cont.)

“Targeted” (also called *orbit*)

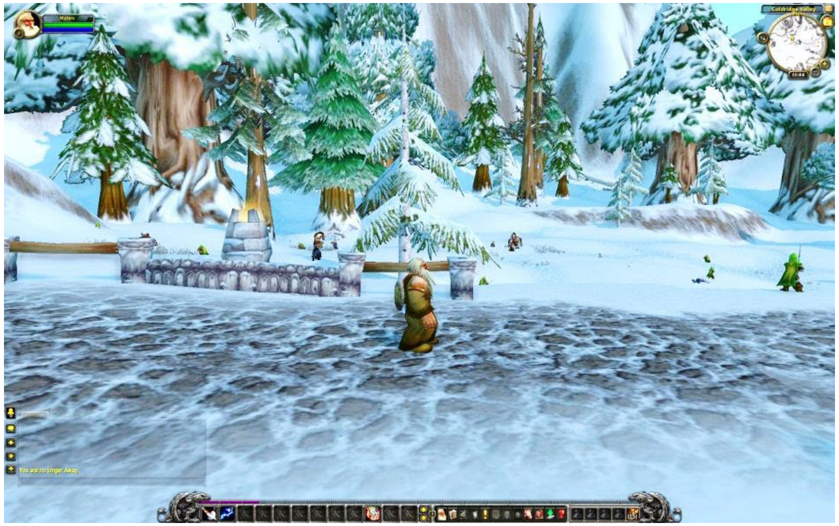
- Camera always looks at avatar
- Camera can be independently controlled in various ways (orbit, zoom)
- In some games, zooming all the way in puts camera in 1P mode.

Example: World of Warcraft



You are no longer Away.

Camera *behind* avatar



You are no longer Away.

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

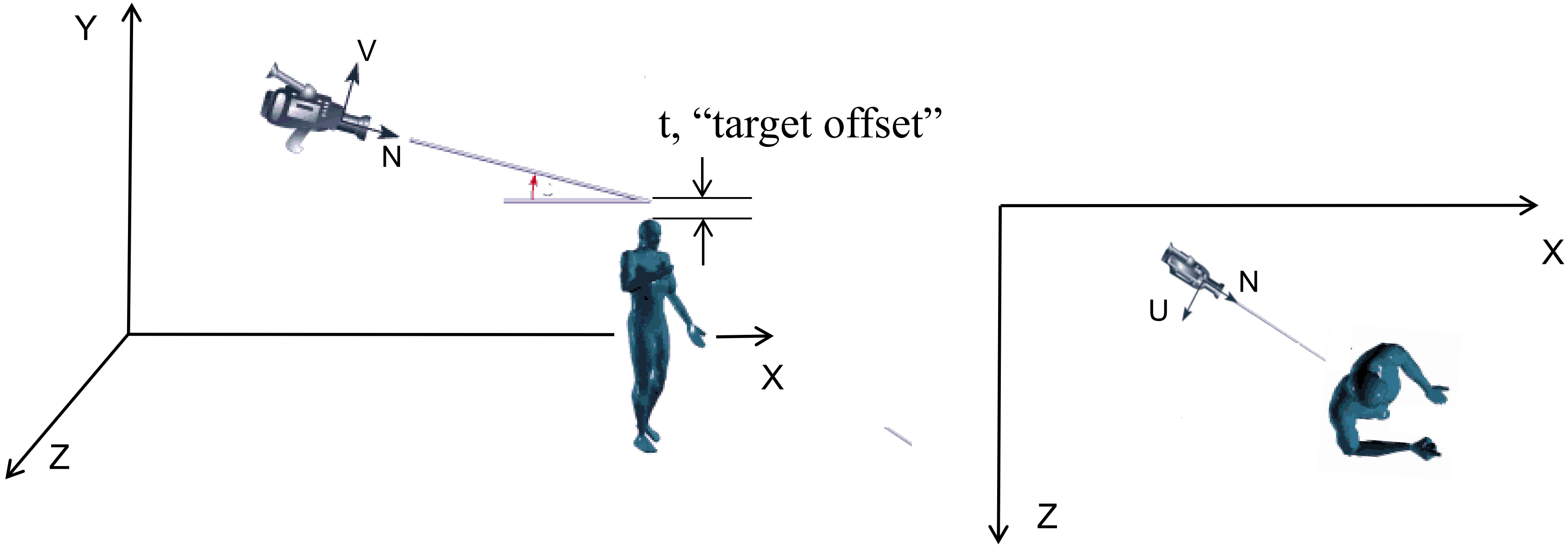
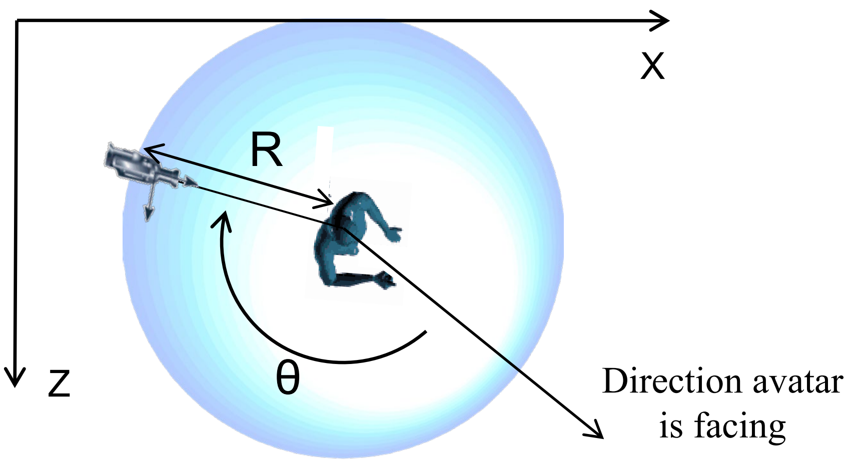


Image credit: www.gamasutra.com

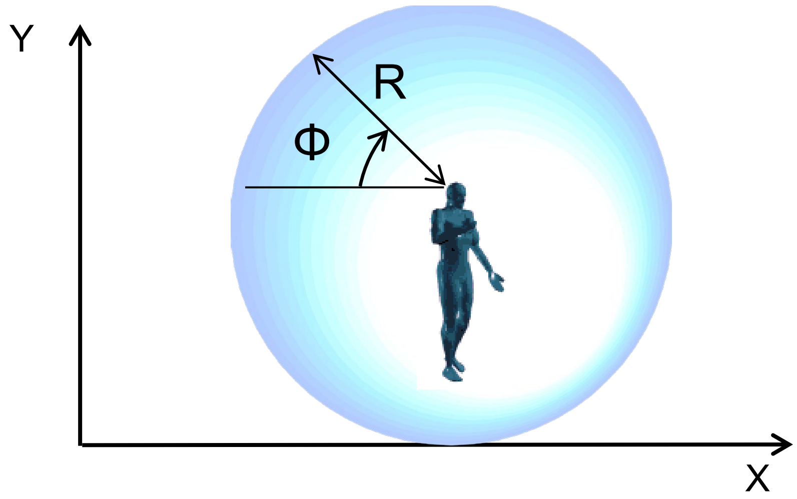
3P Camera Positioning

- Orbit camera position defined in *spherical* coordinates:

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



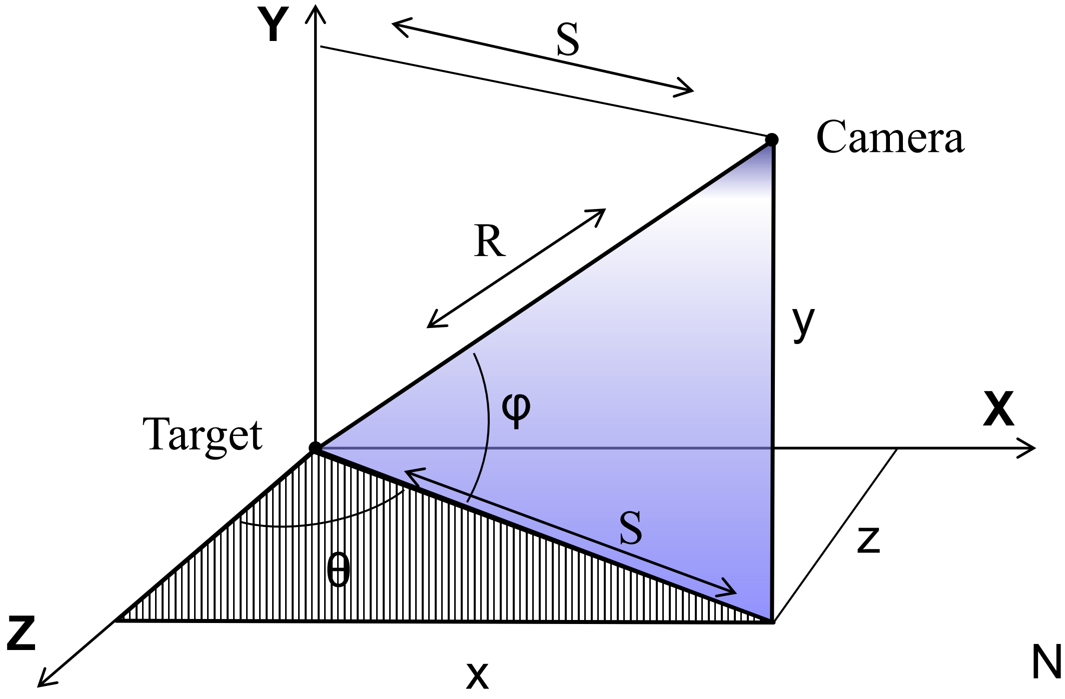
Azimuth



Elevation

Image credit: www.gamasutra.com

Computing Spherical Position



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

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

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

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

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

Note that here x, y, & z are *relative to* *target location*; need to add target location to get camera world position

Targeted Camera Avatar Control

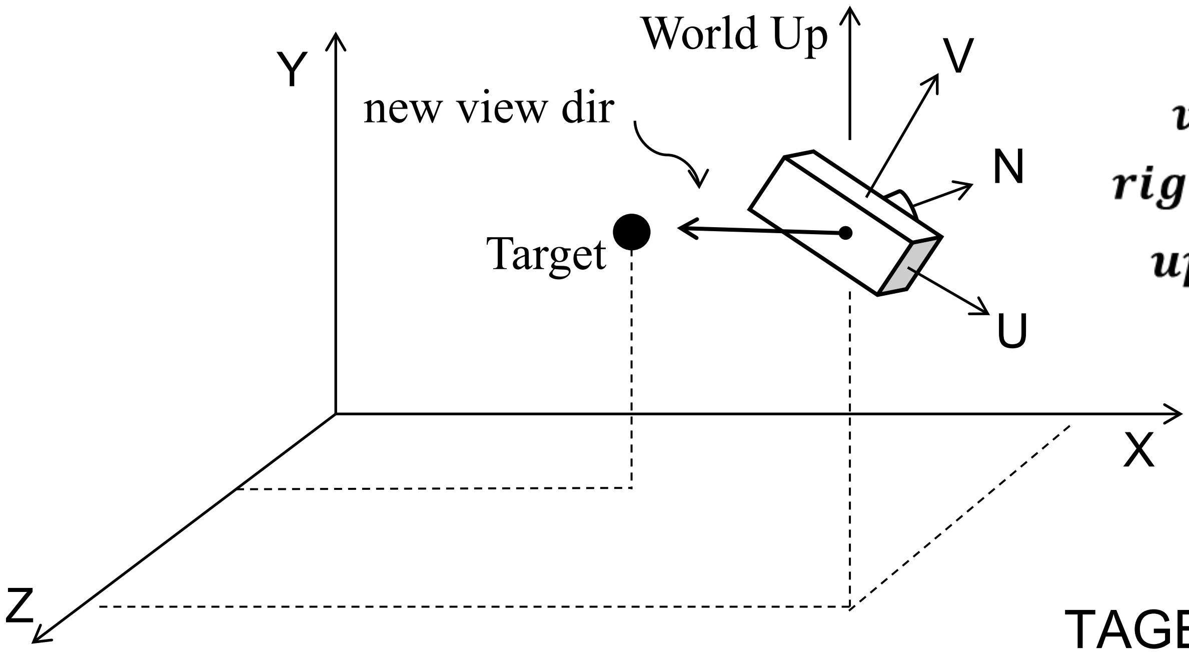
Typical controls:

- 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)

Computing *Look-At*

Given: a camera *position* and *orientation* (U,V,N axis directions)

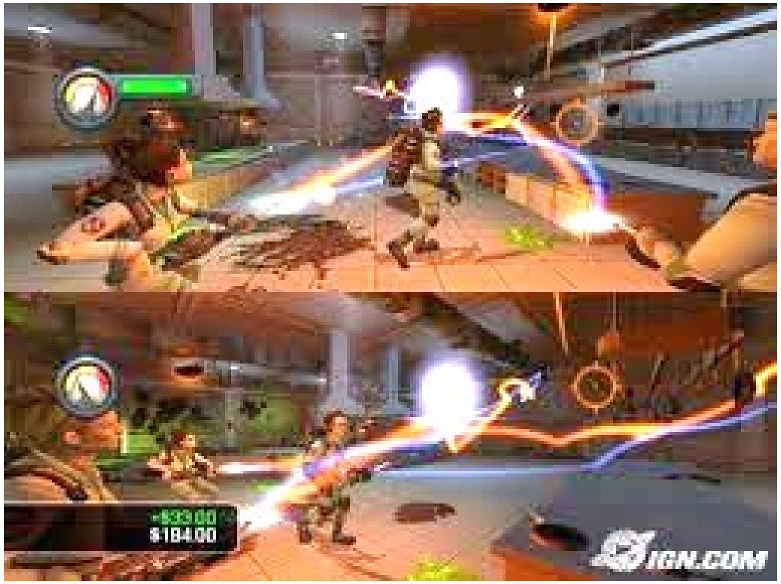
Needed: function **`lookAt(target,worldUp)`**

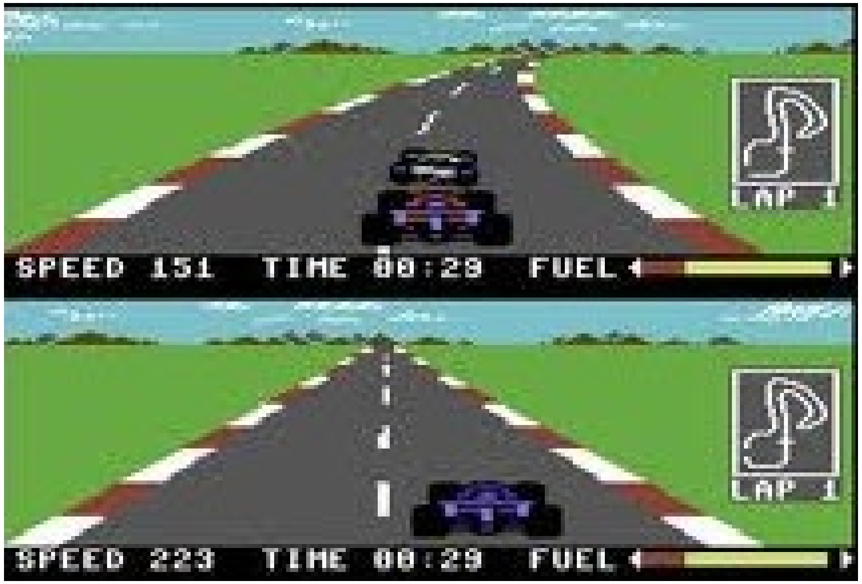


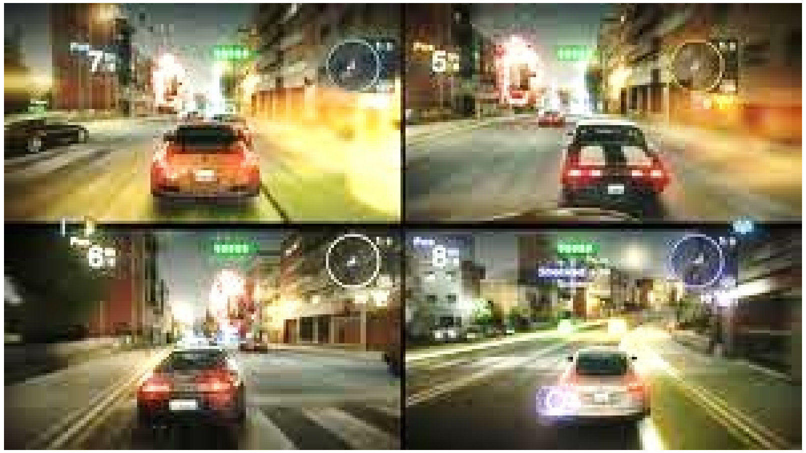
$$viewDir = tPos - cPos$$
$$rightDir = viewDir \times worldUp$$
$$upDir = rightDir \times viewDir$$

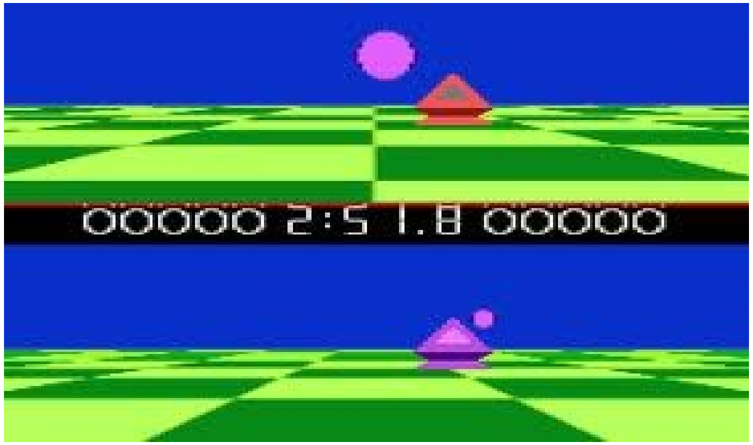
TAGE uses world Y axis for "world up" in its lookAt functions.

Multi-Player “Split-screen”

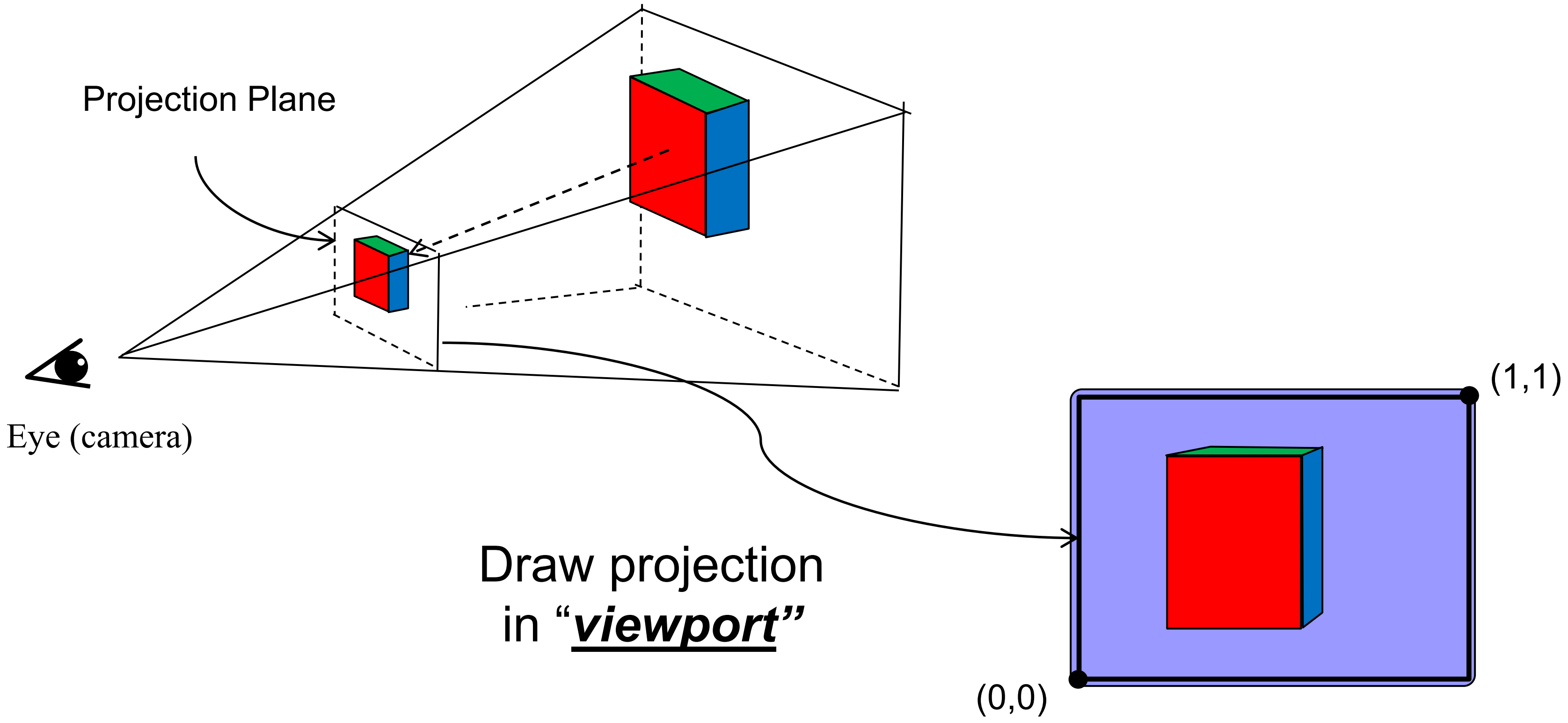








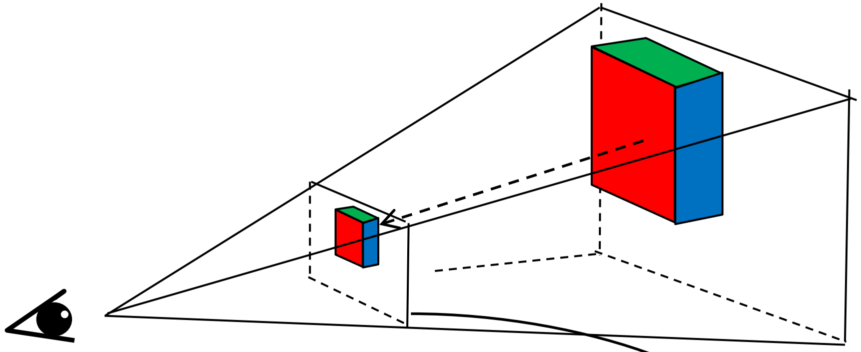
Viewports



Draw projection in "***viewport***"

Default viewport = *entire window*

Setting the Viewport's dimensions

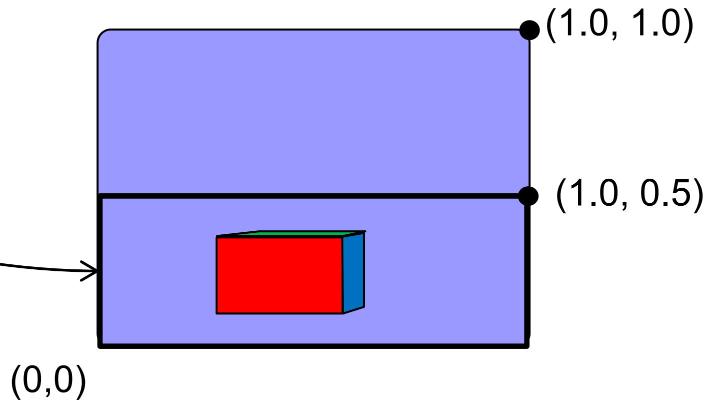


Every viewport has
its own camera

Draw projection in "***viewport***" set by

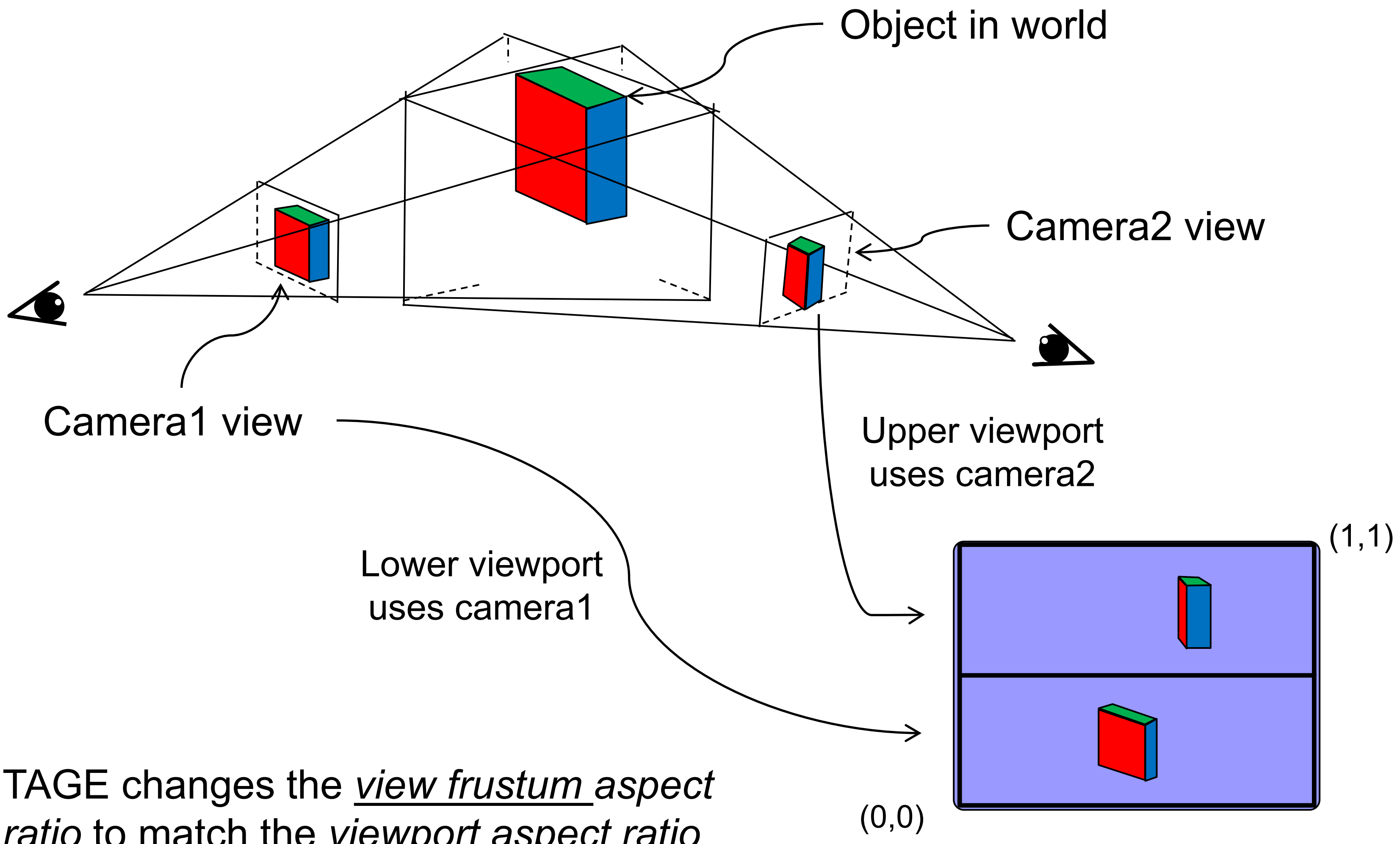
v = new Viewport("LOWER", .01f, .01f, .99f, .49f);

Left, Bottom, Width, Height



Current viewport = *bottom ½ screen*

Multiple Cameras

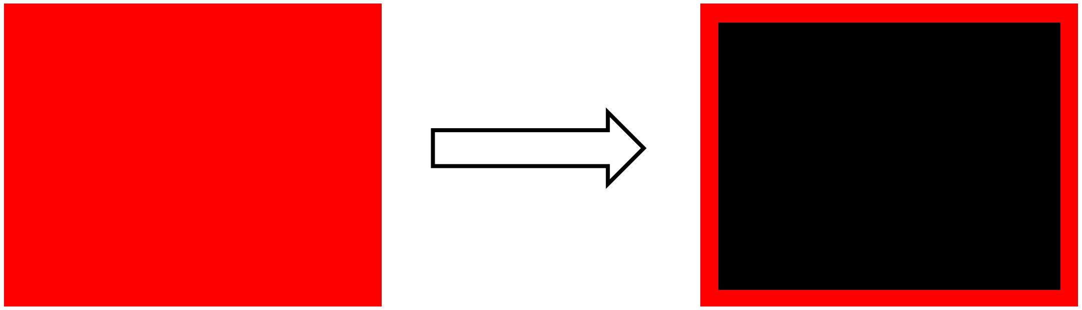


TAGE changes the *view frustum aspect ratio* to match the *viewport aspect ratio*

Creating Viewport Borders

(how the TAGE renderer does it)

1. enables OpenGL “scissor” mode
2. sets scissor size to the entire viewport
3. sets clear color to desired border color
4. clears the viewport (sets entire viewport to border color)
5. sets scissor size to viewport minus border
6. sets clear color to desired viewport background color
7. clears the viewport (sets interior to viewport background color)
8. disables OpenGL “scissor” mode
9. creates OpenGL viewport for interior area



Multiple Cameras (cont.)

TAGE maintains a collection of viewports, each with its own camera.

HUD objects are independent, not tied to a viewport (thus must be placed manually by the programmer).

Creating a viewport automatically creates its camera.

Set the initial configuration of the viewport cameras by overriding ***`createViewports()`***. They can be modified in ***`update()`***. If createViewports() is not overridden, TAGE defaults to one viewport & camera.

The render system draws *each viewport's* view, based on the viewport's camera, in <u>perspective</u> projection.