Camera

Motivation

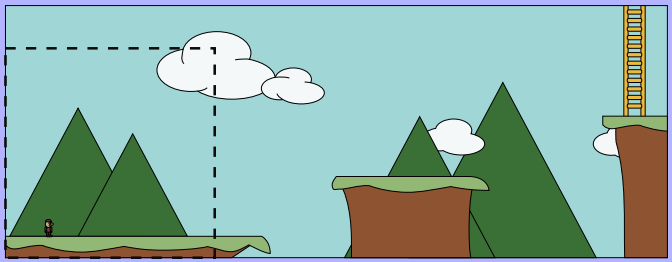
- So far, we've been restricted to a world that fits on the screen
  - -1...1 in x and y
- But: We often want to have a larger virtual space
- We want to be able to handle this

Coords

- We can't change screen space coordinates
  - Always -1...1
- But we *can* move the *world* around!

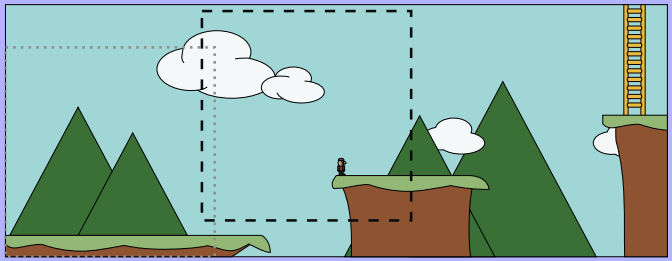
Example

- Suppose we have our virtual world, plus the part that can fit on-screen at once:



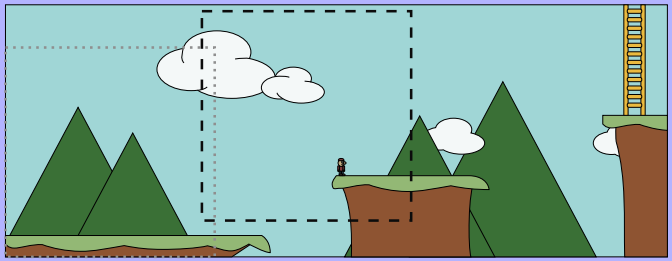
Question

- Suppose we want to move view box right by 0.95 units and up by 0.3 units:



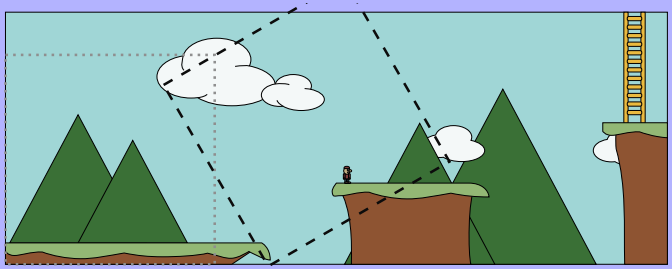
Transform

- Rather than move the camera, we can move the world *left* by 0.95 units and *down* by -0.3 units
- End result is (visibly) the same



View

- But we can do more complex transformations as well
- Suppose we want to spin the viewbox by 30° CCW after moving it:



View

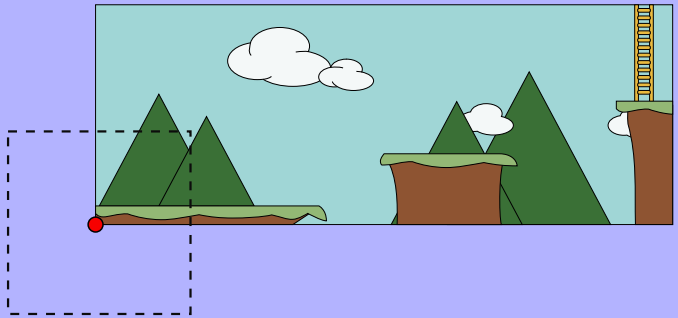
- We know we can accomplish a view spin of 30° CCW by rotating the world 30° CW
- So, in general, our camera transform can be expressed as the product of a rotation matrix R and a translation matrix T
- But: Do we want RT or TR?
  - Does it matter?

Order

- Does it matter?
  - Yes!
- Recall: If we perform a rotation transform, points being rotated appear to orbit the origin
- So let's go back to our virtual world and indicate where (0,0) is...

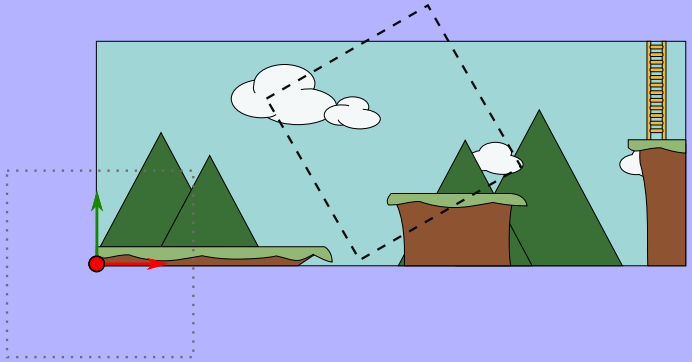
World

- Arbitrarily, let's let the lower left corner of the virtual world be (0,0)
- So with no camera transform, this is our view
  - Remember, window's view is centered on (0,0) with extents -1...1



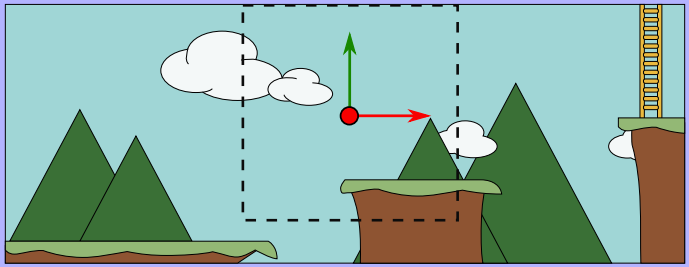
World

- Example: We want to move the camera 3.25 units to the right, 1.3 units up, and then spin 30° CCW



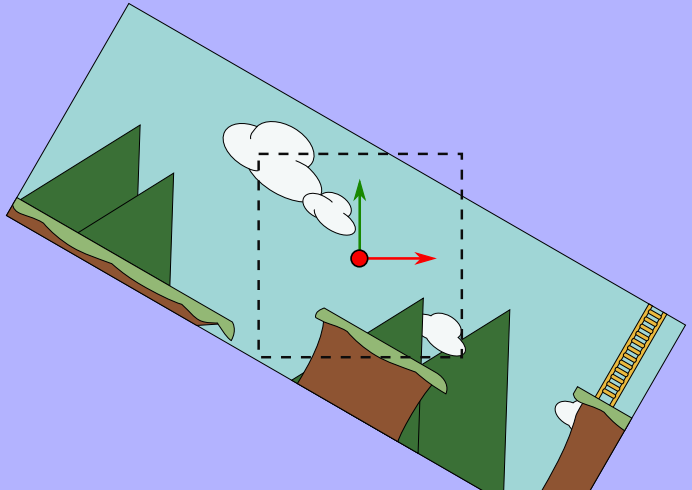
Transforms

- What if we translate, then rotate?
- First, we translate the world by -3.25,-1.3



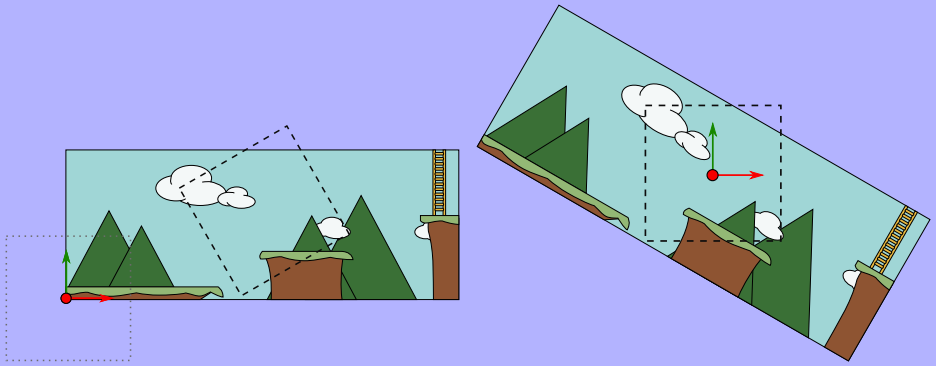
Transforms

- Next, we rotate everything by 30° CW



Compare

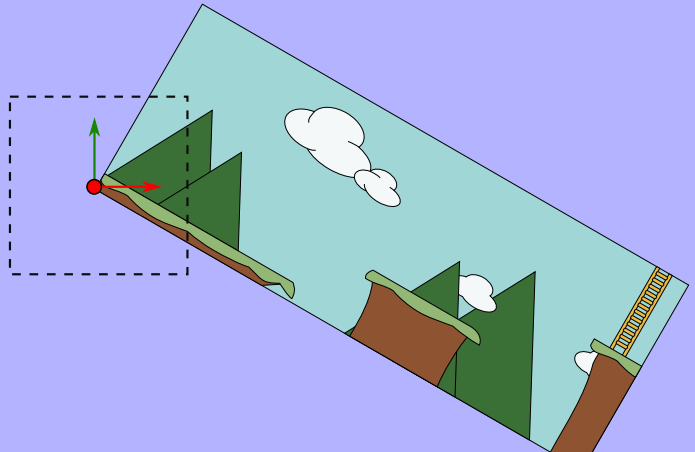
- Compare the result that we wanted with the result we have:



- Perfect!

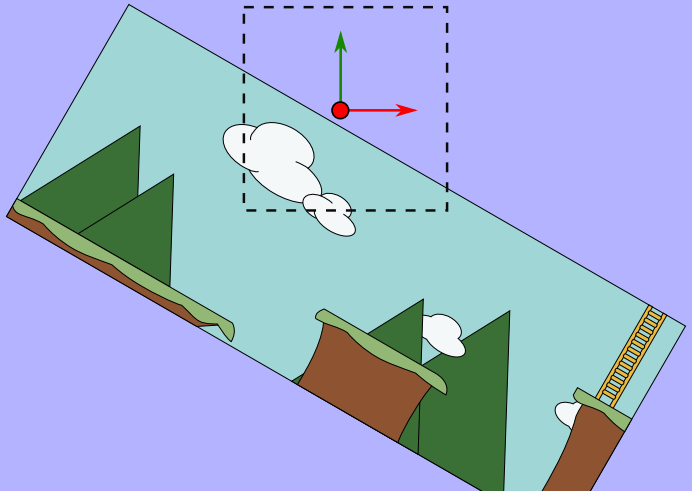
Question

- What if we rotated CW 30° first, then translated by -3.25,-1.3?
- First the rotate:



Then

- Then we translate by -3.25,-1.3:



Result

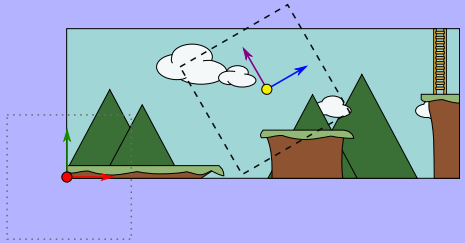
- The two results are very different!
- So: Suppose we are given two pieces of information:
  - A “center of interest” (coi): The 2D world location we want to have in center of screen
  - Head tilt angle: Amount of rotation of camera
- We need to *translate* then *rotate*

Camera

- Sometimes, we want to specify the camera a little differently
- In this case, our camera is specified by:
  - Center of interest (as before)
  - An up vector: A world space vector that is “up” from the camera’s perspective
  - A right vector: A world space vector that points off to the right from the camera’s perspective
  - Note that up and right are perpendicular to each other
    - Strictly speaking, we only *need* one of these
    - We can derive the other one using some vector math
- How can we transform this into a view setup?

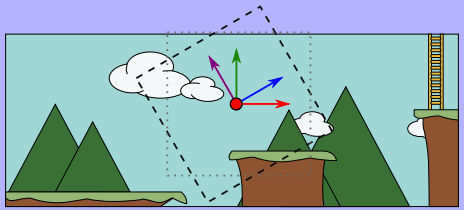
Operation

- Here's what we have as our input:
  - Yellow dot = coi
  - Purple arrow = camera up vector
  - Blue arrow = camera right vector
  - Red dot = origin
  - Red arrow = global X axis
  - Green arrow = global Y axis
- All of these are specified in world space coordinates



Operation

- First, we translate the world by the negative of the coi
- This has the effect of aligning the world origin and the coi point



Rotate

- Next, we must rotate. But how much?
- We could obtain angles via atan2 function
- But there's a faster way...

Matrix

- Create a matrix with the world axes as its rows
  - X axis = (1,0), Y axis = (0,1)
- So we have:

$$W = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- Now create another matrix with the axes of the camera (r=right, u=up)

$$C = \begin{bmatrix} r_x & r_y \\ u_x & u_y \end{bmatrix}$$

- We want a matrix M that will transform C $\rightarrow$ W
  - In other words, W = CM

But...

- Observe: W is just the identity matrix
- So C is the inverse of M
  - Recall: For any matrix Q: $QQ^{-1} = Q^{-1}Q = I$
- There's a handy matrix property: Inverse of an orthogonal matrix Q = Q^T
  - Orthogonal = Row-vectors (and column vectors) are perpendicular to each other
- Thus,

$$M = C^T = \begin{bmatrix} r_x & u_x \\ r_y & u_y \end{bmatrix}$$

Final Matrix

- Final view matrix is TM
  - Where T = $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -c_x & -c_y & 1 \end{bmatrix}$
    - Note: c = center of interest (coi)
  - And M = $\begin{bmatrix} r_x & u_x & 0 \\ r_y & u_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$
    - Where r = right and u = up
    - We add row 0,0,1 and column 0,0,1 to the M matrix so it will be 3x3

Note

- If we wanted, we could premultiply T and M to get:

$$\begin{bmatrix} r_x & u_x & 0 \\ r_y & u_y & 0 \\ -c \cdot r & -c \cdot u & 1 \end{bmatrix}$$

- Notice the dot products in the last row

Implementation

- We normally send camera (or view) matrix to the shader as a uniform
  - We don't combine it with world matrix uniform
  - (Almost) every object has different world matrix, but all objects share same view matrix
  - So we want to "set and forget" view matrix at start of the frame

Example

- We might define a camera class:

```
class Camera:
    def __init__(self,coi):
        self.lookAt( coi, vec2(0,1)  )
    def lookAt(self, coi, up ):
        self.coi = vec3(coi.x,coi.y,1)
        self.up = vec3(up.x, up.y, 0)
        self.right = vec3(self.up.y, -self.up.x, 0)
        self.updateViewMatrix()
    def updateViewMatrix(self):
        self.viewMatrix = mat3(
            self.right.x, self.up.x, 0,
            self.right.y, self.up.y, 0,
            -dot(self.coi,self.right), -dot(self.coi,self.up), 1 )
    def setUniforms(self):
        #assuming shader does p * M and not M * p
        Program.setUniform("viewMatrix",self.viewMatrix)
```

Uniforms

- We have this uniform:
  mat3 viewMatrix;

Shader

- The vertex shader is the one that must use the viewMatrix:

```
(layout location=0) in vec2 position;
...
void main(){
    vec3 p = vec3(position,1.0);
    p = p * worldMatrix;
    p = p * viewMatrix;
    gl_Position = vec4(p.xy,-1,1);
    ...
}
```

Order

- Notice: We must multiply by viewMatrix *after* worldMatrix!
  - Doing it other way around is wrong!
  - Why?

Camera

- We could add a few convenience methods to camera

```
class Camera:
    ...
    def tilt(self,amt):
        M = rotation2( amt )
        self.right = self.right * M
        self.up = self.up * M
        self.updateViewMatrix()
    def pan(self,dx,dy):
        self.coi.x += dx
        self.coi.y += dy
        self.updateViewMatrix()
```

Assignment

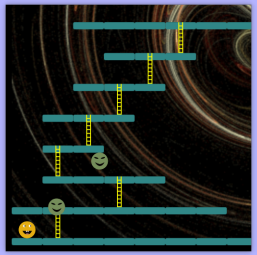
- Retain the existing lab functionality.
- Add a large background image (bigger than the screen)
  - If you are of an astronomical bent, you could look at `https://nssdc.gsfc.nasa.gov/photo_gallery/photogallery-astro-nebula.html` for some freely available images
  - You can omit the starfield, if you like
  - Cite your sources!
- For bonus [+10%], make the background rotate around its own center
  - A “spiral” image might look nice for this
- Continued...

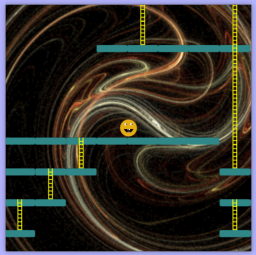
Assignment

- Using the same tile size as before (8 across the screen, 8 down), create a map that's larger than the window both horizontally and vertically
  - You can use this map and tileset if you're not feeling creative .
- Tweak the motion controls: AD=move left/right, WS=move up/down, spacebar fires, return jumps, and (if you have it implemented) x crouches
- Keep the player centered on screen at all times; they can move through the world, which scrolls as needed.
- Continued...

Assignment

- For bonus [+25%], clamp the camera movement
  - This happens when the hero gets near the edge of the world (so the user can't see "past the end" of the world)
  - The screen will be centered on the hero when the hero's not near the world boundaries but the screen won't be centered on the hero when near the world boundaries





Sources

- Chewy Gumball. How to get the rotation matrix to transform between two 3d cartesian coordinate systems? https://gamedev.stackexchange.com/questions/26084/how-to-get-the-rotation-matrix-to-transform-between-two-3d-cartesian-coordinate
- Orthogonal Matrix. https://en.wikipedia.org/wiki/Orthogonal_matrix

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org