CE318: Games Console Programming

Lecture 5: Cameras & Animated Geometries

Philipp Rohlfshagen

prohlf@essex.ac.uk
Office 2.529

Outline

Outline

Camera Types

Last lecture, we looked at transformations (rotations and translations) and different viewing frustums. We are now ready to implement a camera.

There are numerous different camera types, including:

- Static: fixed view of the world (could have different viewpoints)
- Free / first-person: move around freely in 2D or 3D
- Chase: follow a moving object (flexible or rigid)
- Arc-Ball: always looks at one point, moves on surface of 3D sphere

We can use OO-design to re-use the camera implementation in different games. We will only cover the free camera in the lectures and you are encouraged to implement the other camera types in your own time.

Free Moving Camera

A free camera can **move**:

- Up, down, left and right

and can **rotate** around the

- x (pitch) and y (yaw) axes
- Also possible to roll (rotate around z axis)

Any camera that we implement must return a view and projection (which we then use to pass to the draw methods of the objects we intend to render).

So we need:

- View and projection matrices
- A position (i.e., where the camera is at)
- A target and/or direction (i.e., where the camera looks at)
- A definition of up (up and direction can produce right)
- Ways to manipulate the camera: translation and rotations
- Also a speed to regulate how fast the camera moves

Note: free camera implementation based on A. Reed, chapter 11.

Free Moving Camera

First we define our class variables and properties:

```
public Matrix View { get; private set; }
public Matrix Proj { get; private set; }

float angleOfView, nearPlane, farPlane;
Vector3 position, target, up, look;
const float speed = 0.3f;
```

The constructor takes all the initial values required and initialises everything:

```
public CameraAngle(Game game, Vector3 position, Vector3 target,
     Vector3 up, float angleOfView, float nearPlane, float farPlane) :
      base(game)
{
  this.position = position;
  this.up = up;
  this.target = target;
  this.angleOfView = angleOfView;
  this.nearPlane = nearPlane;
  this.farPlane = farPlane;
}
```

We use the `Initialize()` method to set up everything:

```
public override void Initialize()
{
  look = target - position;
  look.Normalize();

  SetUpProj();
  SetUpView();
}
```

Free Moving Camera

We use two helper methods to initialise view and projection:

```
public void SetUpProj()
{
  Proj = Matrix.CreatePerspectiveFieldOfView(angleOfView, Game.
       GraphicsDevice.Viewport.AspectRatio, nearPlane, farPlane);
}
```

```
public void SetUpView()
{
  View = Matrix.CreateLookAt(position, position + look, up);
}
```

This essentially creates a static camera we could aready use for our game. All we need to do now is to add an update method to manipulate the vectors according to user input. We will use a keyboard but you should try to use a game pad in the labs.

We override the update method in `GameComponent` and place the code in there:

```
public override void Update(GameTime gameTime)
{

}
```

Free Moving Camera

First we tackle translation which is very simple.

```
//forward
if (Game1.input.IsKeyDown(Keys.W))
  position += look * speed;
//backward
if (Game1.input.IsKeyDown(Keys.S))
  position -= look * speed;
//right
if (Game1.input.IsKeyDown(Keys.D))
  position += Vector3.Cross(look, up) * speed;
//left
if (Game1.input.IsKeyDown(Keys.A))
  position -= Vector3.Cross(look, up) * speed;
//up
if (Game1.input.IsKeyDown(Keys.P))
  position += up * speed;
//down
if (Game1.input.IsKeyDown(Keys.L))
  position -= up * speed;
```

This code simply alters the position of the camera given user input.

Note: only the position is altered. The vectors `look` and `up` remain the same and hence we keep looking in the same direction at all times.

Q Is the direction the same as the camera's target?

Free Moving Camera

Translation relies on the camera's local coordinate system: look (forward), right and up. These vectors are manipulated by the rotations we will perform.

Class exercise: which vectors are affected in case of the following?

- pitch
- yaw
- roll

Q Is it necessary to always adjust all of them?

It is important to plan how you want your camera to behave. This has a huge impact on game play and even if the camera is "technically correct", it may be the wrong choice for the game.

The following implemention is just one example of what can be done. Feel free to modify the camera to suit your needs.

Free Moving Camera

Next we do the rotations of yaw and roll.

```
//right (around y)
if (Game1.input.IsKeyDown(Keys.I))
{
  look = Vector3.Transform(look, Matrix.CreateFromAxisAngle(up, -
       MathHelper.ToRadians(1)));
}
//left (around y)
if (Game1.input.IsKeyDown(Keys.J))
{
  look = Vector3.Transform(look, Matrix.CreateFromAxisAngle(up,
       MathHelper.ToRadians(1)));
}
//roll right (around z)
if (Game1.input.IsKeyDown(Keys.U))
{
  up = Vector3.Transform(up, Matrix.CreateFromAxisAngle(look, -
       MathHelper.ToRadians(1)));
}
//roll left (around z)
if (Game1.input.IsKeyDown(Keys.H))
{
  up = Vector3.Transform(up, Matrix.CreateFromAxisAngle(look,
       MathHelper.ToRadians(1)));
}
```

Yaw: only direction changes.
Roll: only up changes.

Free Moving Camera

Finally we implement pitch (direction and up change):

```
//up (around x)
if (Game1.input.IsKeyDown(Keys.O))
{
  look = Vector3.Transform(look, Matrix.CreateFromAxisAngle(Vector3.
       Cross(up, look), MathHelper.ToRadians(1)));
  up = Vector3.Transform(up, Matrix.CreateFromAxisAngle(Vector3.Cross(
       up, look), MathHelper.ToRadians(1)));
}
//down (around x)
if (Game1.input.IsKeyDown(Keys.K))
{
  look = Vector3.Transform(look, Matrix.CreateFromAxisAngle(Vector3.
       Cross(up, look), -MathHelper.ToRadians(1)));
  up = Vector3.Transform(up, Matrix.CreateFromAxisAngle(Vector3.Cross(
       up, look), -MathHelper.ToRadians(1)));
}
```

At the end of the update method, we must call: `SetUpView()`.

That is it. We can now use this camera to explore the 3D world.

Note: doing rotations can get confusing. You should always ask yourself which axis you need to rotate around and which vectors will be affected.

You should aways have an orthogonal matrix for the camera at all times.

Outline

Rotating Boxes

We have covered in depth how translations and rotations are performed in XNA. It is now time to put that theory to the test. First, we will animate some boxes (models) and then implement a robotic arm with hierarchical transformations.

We will look at the following:

- Rotate an object in place at origin
- Rotate an object in place away from origin
- Rotate an object around an arbitrary point
- Combine the individual transforms into a single transform

We assume each box has a translation and rotation matrix. Rotating an object at the origin is easy: there is no translation so simply apply one or more of the following:

- Matrix.CreateRotationX()
- Matrix.CreateRotationX()
- Matrix.CreateRotationX()
- Matrix.CreateFromAxisAngle()

Rotating Boxes

Now, if we want to rotate an object away from the origin, we need translation and rotation. We can do one of the following:

```
world = rotation * translation;
```

```
world = translation * rotation;
```

Q What is the difference in outcome between the two?
Q What if the object is already translated?

Given the different behaviours, it begs the questions how to combine these transformations into a single joint action. First, how to rotate around an arbitrary point?

Assume a pivot point `Vector3 pivot`, then:

```
world = translation * Matrix.CreateTranslation(-pivot) * rotation *
     Matrix.CreateTranslation(pivot);
```

What is happening? Move to pivot and rotate in place, then move the rotation to actual position.

Rotating Boxes

Finally, we want to rotate boxes in place away from the origin and simultaneously rotate them around some other boxes which also rotate in place.

```
world1 = rots1 * trans1;
world2 = rots2 * trans2;

world3 = rots3 * Matrix.CreateTranslation(pos3 - pos1) * rots3 *
     trans1;
world4 = rots4 * Matrix.CreateTranslation(pos4 - pos2) * rots4 *
     trans2;
```

What is happening here? We rotate boxes 3 and 4 in place at the positions where boxes 1 and 2 are, then move them up a bit and rotate them around their pivots (i.e., boxes 1 and 2):

rotate in place * up from pivot * rotate around pivot * put into position

Quick Note

In the demo, you can see the boxes, some lines and text to indicate which mode we are in. The text needs to be drawn using the `spriteBatch`. If you mix 2D and 3D graphics, the 3D graphics may no longer display correctly.

The `SpriteBatch` changes several graphic device states:

- GraphicsDevice.BlendState = BlendState.AlphaBlend;
- GraphicsDevice.DepthStencilState = DepthStencilState.None;
- GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;
- GraphicsDevice.SamplerStates[0] = SamplerState.LinearClamp;

`SpriteBatch` also modifies the buffers and applies its own effect. So, before drawing 3D, it might be neccessary to reset these states:

- GraphicsDevice.BlendState = BlendState.Opaque;
- GraphicsDevice.DepthStencilState = DepthStencilState.Default;
- GraphicsDevice.SamplerStates[0] = SamplerState.LinearWrap;

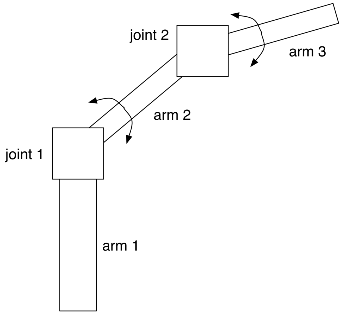
Based on a blog by Shawn Hargreaves (links on course web page)

A Robotic Arm

So far, we have used single-mesh objects and scaled, rotated and translated them. We have also seen an example of a 2-mesh object (2 boxes) and how one mesh was rotated independently of the other. Such animated objects are found commonly in games. To create an animated object, it is essential to have a good understanding of how local translations and rotations affect the overall position of all meshes of the model.

We will create a simple 3D articulated arm (ARM) with *n* **arms** and $n-1$ **joints** as shown on the right.

Each of the two arms may be controlled independently. The other parts must behave accordingly.



A Robotic Arm

The ARM will be constructed and controlled as follows: we have a set of arms that can be manipulated using keys. We also have a set of joints for visualisation only. We can do the following:

- Activate a section of the arm using *tab*
- Rotate the base around the y-axis
- Rotate any of the arms around the x-axis

For any rotation made, all parts of the ARM "above" the arm manipulated need to be transformed also. We thus need to recursively accumulate all rotations and translations and apply them appropriately to display each arm correctly.

We can treate 'arm 1' (the base) as the **root**, with child 'arm 2' which in turn has as child 'arm 3' and so on. We will draw the ARM using primitives.

A Robotic Arm

We can use a generic cube generator that draws cubes with or without base or arbitrary dimensions. Each cube will be centred at the origin. We create n closed cubes for the arms and $n-1$ open cubes for the joints. The dimensions are up to you. We turn culling off for this exercise.

We create a class called `Arm` where we construct and control the ARM. We have the following class variables:

```
const int numParts = 4;
const float height = 4;
const float width = 0.5f;

GraphicsDevice device;
Cube[] arms;
Cube joint;

Matrix[] rotations;
Matrix global = Matrix.Identity;

Vector3 dimension = new Vector3(width, height, width);

int currentActive;
float angle = MathHelper.ToRadians(1);
```

A Robotic Arm

We initialise everything in `Init()`:

```
public void Init()
{
  arms = new Cube[numParts];
  rotations = new Matrix[numParts];

  for (int i = 0; i < arms.Length; i++)
  {
    arms[i] = new Cube(true, dimension, device);
    rotations[i] = Matrix.Identity;
  }

  joint = new Cube(false, new Vector3(2 * width, 2 * width, 2 * width)
       , device);

  currentActive = 0;
  arms[currentActive].activate();
}
```

This creates the arms and initiliases the rotations (all identity at the beginning). We only use one joint and simply draw it in different places (could do the same with the arms if we keep track of which arm is active externally).

Whenever space is pressed, we change the active arm. The active arm has a white outline and reacts to key-input (direction keys).

A Robotic Arm

The key input is processed in `Update()`:

```
public void Update()
{
  if (currentActive == 0)
  {
    if (Keyboard.GetState().IsKeyDown(Keys.Left))
      rotations[0] *= Matrix.CreateRotationY(angle);
    if (Keyboard.GetState().IsKeyDown(Keys.Right))
      rotations[0] *= Matrix.CreateRotationY(-angle);
  }
  else
  {
    if (Keyboard.GetState().IsKeyDown(Keys.Up))
      rotations[currentActive] *= Matrix.CreateRotationX(angle);
    if (Keyboard.GetState().IsKeyDown(Keys.Down))
      rotations[currentActive] *= Matrix.CreateRotationX(-angle);
  }

  if (Keyboard.GetState().IsKeyDown(Keys.W))
    global *= Matrix.CreateTranslation(Vector3.Forward);
  if (Keyboard.GetState().IsKeyDown(Keys.S))
    global *= Matrix.CreateTranslation(Vector3.Backward);
  if (Keyboard.GetState().IsKeyDown(Keys.A))
    global *= Matrix.CreateTranslation(Vector3.Left);
  if (Keyboard.GetState().IsKeyDown(Keys.D))
    global *= Matrix.CreateTranslation(Vector3.Right);
}
```

Updating the active arm is done in `Game1`.

A Robotic Arm

All the transformations are computed in the `Draw()` method. For each part, we need to obtain its position by accumulating the transforms of all its parents. We start at the root and work our way up.

```
public void Draw(Matrix view, Matrix proj)
{
  Matrix transform = Matrix.Identity;

  for (int i = 0; i < numParts; i++)
  {
    Matrix moveDown = Matrix.CreateTranslation(0, -height / 2, 0);
    Matrix moveUp = Matrix.CreateTranslation(0, height, 0);

    transform = moveUp * rotations[i] * transform;
    arms[i].Draw(moveDown * transform * global, view, proj);

    if (i < numParts - 1)
      joint.Draw(transform * global, view, proj);
  }
}
```

Note: we move the arm down, apply the transform and then move it into its proper place. This ensures we rotate around the correct pivot point (e.g., 'arm 2' rotates around top of 'arm 1' which, thanks to the move down, was at the origin).

Outline

Assessment

As part of the assessment for CE318, you will need to complete an assignment worth 20%. The task is to develop a 3D game of your choice. You may freely choose a game of your liking but the game must include:

- A moving object.
- An appropriate camera implementation.
- Collision detection amongst all elements in the game.
- Collecting items such as health packs, boosters etc.
- Allow for different levels that are loaded from a file.
- AI opponents with simple yet better-than-random behaviours.

Your game should require simple rules that make the game interesting and, of course, your game should be fun to play. It is better to implement something simpler well than something more complicated not so well. But don't make your game too simple!

Assessment

You may use the code we developed throughout the lectures and labs. For higher marks, you should aim to optimise gameplay and to visually improve the game. Suggestions for advanced features include:

- Different types of opponents.
- More advanced opponent AI.
- Different types of objects in the game (for collection, collisions).
- Articulated objects in the game.
- More advanced visual features such as particle effects, billboarding, etc.
- Multiplayer options (split-screen).

Detailed instructions may be found in the assignment description which is now available on the course webpage. Some suggestions for suitable games include: 3D shooters, extended 3D Pong with object collection, extended 3D Angry Birds, racing games, spaceship simulators, etc.

Outline

Summary & Lab Preview

In this half lecture we looked at camera implementations and implemented a free camera suitable for flexible movement in 3D space. We also looked at animated objects and constructed a simple articulated robotic arm using primitives.

We covered

- Camera types
- Free camera
- Animated objects
- Articulated robotic arm

We also discussed the assignment for CE318.

In the lab you will implement an articulated robotic arm using a multi-mesh model instead of primitives. You should place this model into your 3D world together with the other animated primitive types / simple models you have added over the last few labs. Also, you should complete the free camera provided to accommodate yaw and roll.