Smooth Follow Camera for 2.5D Platformers

Marcelo Laborda

Smooth Follow Camera for 2.5D Platformers

Cameras are undeniably one of the most important components of a video game. They are the only device that can show the game's world to the player, and they are tightly related to game aesthetics. A camera can help communicate the game's story, guide the player's eye movement towards a desired focal point, direct the player's movement within the game space, or even make the player more emotionally involved and evoke different emotions. But most importantly, cameras are part of the foundation of gameplay; they dictate how the player interacts with the game. The camera is a powerful device that must not be neglected, as it could affect the gameplay negatively if done wrong. Consequently, it is of vital importance to use a functional and robust camera design when developing a video game.

This tutorial provides a step-by-step guide to designing and implementing a simple but effective camera for 2.5D platform games using *Unity* version 4.6.2 and scripting with *C#* in *MonoDevelop*. A 2.5D, or two-and-a-half dimensional camera model, features 3D graphics, but its gameplay is restricted to a two-dimensional plane. In this tutorial, it is assumed that the reader has set up a scene with a character controller, a camera, and different platforms to jump to.

1 - Tracking the Character's Movement

There is one thing that all camera systems in platformers have in common: they all follow the character's movement. By doing this, the player is always aware of the character's presence and is able to observe its surroundings without losing sight of the action.

A simple approach would be to attach to the camera a script that assigns the position of the playable character to the camera's transform position, or to make the camera a child of the playable character. This, however, makes the camera movement seem unnatural, with sudden stops and abrupt transitions. A better approach is to make the camera follow the player with a slight delay and reaction time. This allows the player to anticipate the action. The principle of anticipation in animation is the preparation of an action and is a fundamental element in creating natural motion.

To make a camera with this behavior, create a new script and place it on the main camera. Edit the new script in *MonoDevelop*, and declare the following variables:

```
public Transform target;

private Vector3 currentPos = Vector3.zero;
private Vector3 targetPos = Vector3.zero;
```

Target references the game object that the camera will follow. This variable is set to *public* in order to expose a reference to other objects. In this tutorial, the main camera follows the player, so the target is the character controller in the scene. To get its *Transform* component, drag a character controller on the *target* slot in the inspector.

The variable *currentPos* stores the camera's current position, while *targetPos* stores the position the camera is trying to reach. To make the camera gradually change its position towards a desired target over time, use the method *Vector3.SmoothDamp*. Add the following variable declarations to your script, and include the *LateUpdate* function:

```
public float smoothTime = 0.4f;
public float maxSpeed = 100.0f;
public float zOffset = -10.0f;
private Vector3 currentVelocity = Vector3.zero;

private void LateUpdate(){
      if (target != null) {
            targetPos = new Vector3 (target.transform.position.x,
            target.transform.position.y,
            target.transform.position.z + zOffset);
      }
      currentPos = Vector3.SmoothDamp(currentPos, targetPos, ref
      currentVelocity, smoothTime, maxSpeed, Time.deltaTime);

      transform.position = currentPos;
}
```

In this script, the target position is always the point in space where the character is situated. The camera's position is constantly updated to an interpolated value between its current position and the target position.

LateUpdate is called once per frame after all *Update* functions complete their execution. It is recommended to use *LateUpdate* instead of *Update*, since the game character is moved inside the *Update* function and the camera needs to track its position. The variable *smoothTime* represents the time the camera will take to reach the target. *MaxSpeed* is used to limit the camera's speed. *ZOffset* determines the distance between the camera and the player on the z-axis. *CurrentVelocity* is necessary to store the current velocity of the operation, and it is modified by the function automatically.

2 - Adding a Camera Offset

In most action-based platformers, such as *Outland* or *Guacamelee!,* to name a few, the main camera automatically centers the character in the shot (see Figure 1). This particular area, located in the center of the screen, is where the primary action of the game potentially occurs, where players focus their attention. The previous script is perfectly suitable for these types of games. In most non-combat platformers, however, the camera is slightly off-centered. This allows the player to focus on the space in front of the character. In these games, the player is looking at the different platforms, collectibles, or enemies that appear in front of the character because they require quicker reactions and better timing. Figure 2 shows an example of an offset camera in a 2.5D platformer.

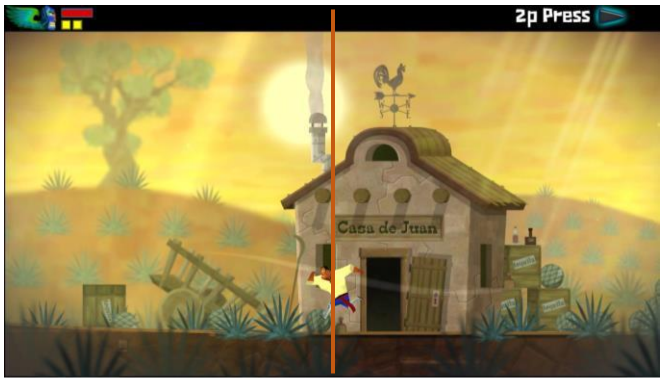

Fig. 1. This camera centers the character in the shot

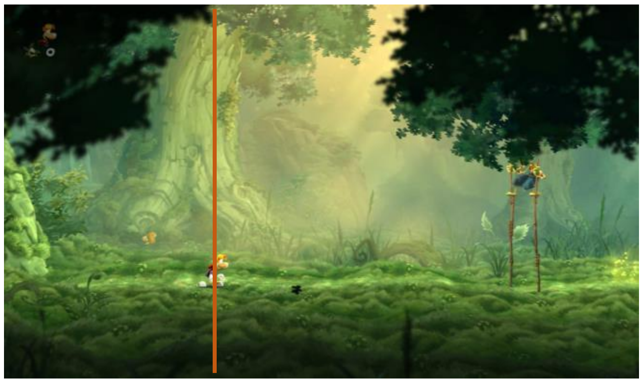
Fig. 2. This camera shows what is ahead

To add this behavior to the main camera, update the script created in the previous section adding the following variable declaration:

```
public float xOffset = 3.0f;
```

This variable represents how far the camera is from the character to provide a better field of vision. The higher the value, the bigger the distance between the character and the camera.

To make this work, modify the line:

```
targetPos = new Vector3 (target.transform.position.x,
target.transform.position.y, target.transform.position.z +
zOffset);
```

To be

```
targetPos = new Vector3 (target.transform.position.x + xOffset *
Mathf.Sign(target.transform.localScale.x),
target.transform.position.y, target.transform.position.z +
zOffset);
```

This way, the target position will always be ahead of the character – regardless of the direction the character is facing – revealing more of the surrounding space.

Mathf.Sign is used to determine which way the player is currently facing. It is important to note that this will only work if the character controller flips horizontally when changing direction. (e.g. when using “`transform.localScale *= -1;`”).

3 - Restricting Camera Movements

Since the camera always follows the character's movement, if the player moves too far to one side of the scene, the edges of the background become visible. That means the only solution is to assign constraints to the camera to limit its movement inside the level bounds.

Use the class *Rect* to define the boundaries of the level:

```
public Rect levelBounds = new Rect(0.0f, 0.0f, 10.0f, 10.0f);
```

In this particular example, the rectangle is positioned at the world origin, and is 10 units long and 10 units wide. Once the variable is defined, add the following method:

```
private Vector3 ConstraintLevelBounds(Vector3 targetPos){
      Vector3 viewportSizeBy2 = ViewportSizeBy2 ();

      float posX = Mathf.Clamp(targetPos.x, levelBounds.xMin +
      viewportSizeBy2.x, levelBounds.xMax - viewportSizeBy2.x);

      float posY = Mathf.Clamp(targetPos.y, levelBounds.yMin +
      viewportSizeBy2.y, levelBounds.yMax - viewportSizeBy2.y);

      float posZ = targetPos.z;

      return new Vector3(posX, posY, posZ);
}

private Vector3 ViewportSizeBy2(){
      return (camera.ViewportToWorldPoint(
      new Vector3 (1, 1, -camera.transform.position.z)) -
      camera.ViewportToWorldPoint (
      new Vector3 (0.5f, 0.5f, -camera.transform.position.z)));
}
```

ConstraintLevelBounds will clamp the value of the target position if the camera viewport is outside the boundaries of the level. *ViewportSizeBy2* determines the distance in world units between the center of the viewport and the top-right corner to make sure the camera stays inside the level at all times.

To change the size and position of the rectangle that represents the boundaries of the level, set new values on the inspector. If the size of the level in units is unknown, use the function *OnDrawGizmos* and add the following code:

```
void OnDrawGizmos() {

      Gizmos.color = Color.blue;

      Gizmos.DrawLine(
      new Vector3(levelBounds.xMin,levelBounds.yMax, 0.0f),
      new Vector3(levelBounds.xMin,levelBounds.yMin,0.0f));

      Gizmos.DrawLine(
      new Vector3(levelBounds.xMin,levelBounds.yMin,0.0f),
      new Vector3(levelBounds.xMax,levelBounds.yMin,0.0f));

      Gizmos.DrawLine(
      new Vector3(levelBounds.xMax,levelBounds.yMin,0.0f),
      new Vector3(levelBounds.xMax,levelBounds.yMax,0.0f));

      Gizmos.DrawLine(
      new Vector3(levelBounds.xMax,levelBounds.yMax,0.0f),
      new Vector3(levelBounds.xMin,levelBounds.yMax,0.0f));
}
```

This will draw a custom gizmo to give visual aids on the viewport when in edit mode. Lastly, save the script and run the scene. Test different values to find the most adequate camera for the game's needs.

References

Unity Manual. (n.d.). Retrieved February 6, 2015, from http://docs.unity3d.com/

2D Camera "Smooth Follow" (2012, October 11). Retrieved February 6, 2015, from http://answers.unity3d.com/questions/29183/2d-camera-smooth-follow.html