3C and camera

The goal of this course is to setup basic 3D cameras and learn the 3D geometry you need to achieve that.

You will study :

- A rotating spring arm camera in unity
- A dead zone for the camera to follow
- A transition between third person and external cinematic camera
- Transition to a first person camera

Orbiting camera with a spring arm

Orbiting camera

We want a camera that will rotate around the character.

Create a SpringArm game object. Setup a SpringArm script on it. Set the MainCamera as a child of it.

We will first handle the orbiting feature. Here will be the code of the Update function:

```
void Update()
{
    // If target is null, get out
    if(!target)
        return;

    // Handle mouse inputs for rotations
    if (useControlRotation && Application.isPlaying)
        Rotate();
    }
}
```

We will need some variables and a Rotate function.

```
#region Rotation Settings

[Space]
[Header("Rotation Settings \n-----------------------")]
[Space]
[SerializeField] private bool useControlRotation = true;
[SerializeField] private float mouseSensitivity = 500f;

// For mouse inputs
private float pitch;
private float yaw;

#endregion
```

The first variable is a setting for our camera, and the others will handle the rotation:

```
/// <summary>
/// Handle rotations
/// </summary>
private void Rotate()
{
    // Increment yaw by Mouse X input
    yaw += Input.GetAxisRaw("Mouse X") * mouseSensitivity * Time.deltaTime;
    // Decrement pitch by Mouse Y input
    pitch -= Input.GetAxisRaw("Mouse Y") * mouseSensitivity * Time.deltaTime;
    // Clamp pitch so that we can't invert the the gameobject by mistake
```

```
        pitch = Mathf.Clamp(pitch, -90f, 90f);

        // Set the rotation to new rotation
        transform.localRotation = Quaternion.Euler(pitch, yaw, 0f);
    }
```

The final rotation in a 3D game is most game engines is a quaternion. This mathematical structure can be applied to vectors or to other quaternions to make them rotate around the 3 dimensional axes.

Here, we want to define a final rotation in function of the mouse movement. We calculate the quantity of move during last frame, multiply it by a mouse sensivity factor and delta-time. We clamp the pitch to avoid the camera to go to high or to low. Then we can create our quaternion, and set the local rotation with it.

You should have an orbiting camera. But it rotates around nothing! We have to make it follow our character.

Following camera

We will need some variable to setup the follow-up behaviour. Among them is a reference to the transform of the game object we want to follow.

```
    #region Follow Settings

    [Space]
    [Header("Follow Settings \n--------------------")]
    [Space]
    [SerializeField] private Transform target;
    [SerializeField] private float movementSmoothTime = 0.2f;
    [SerializeField] private Vector3 targetOffset = new Vector3(0, 1.8f, 0);

    // refs for SmoothDamping
    private Vector3 moveVelocity;

    #endregion
```

Unity provides us with a SmoothDamp function we can use to make the camera smoothly follow the character.

```
    void Update()
    {
        // If target is null, return from here: NullReference check
        if(!target)
            return;

        // Handle mouse inputs for rotations
        if (useControlRotation && Application.isPlaying)
            Rotate();

        // Follow the target applying targetOffset
        Vector3 targetPosition = target.position + targetOffset;
        transform.position = Vector3.SmoothDamp(transform.position,
            targetPosition, ref moveVelocity, movementSmoothTime);
    }
```

What is exactly this SmoothDamp function? Well, we don't have access to the unity code, but with some internet search we can localize the source of the SmoothDamp function (Game Programming Gems 4, Chapter 1.10, cf. attached document). Basically it eases in and out the move from a current position to a target position, taking into account and changing a move velocity, with a time parameter.

Now the camera will follow the player with a nice smooth damp.

Setting an arm length

A spring arm is, in a video game, a classic way to describe the behaviour of a following camera in a 3rd person view.

The "arm" part means there is a constant length and orientation that should be respected. The "spring" part refers both to the damped move when the camera follow the target and to the fact that this camera must move along the arm to avoid collisions.

We will start by providing the arm feature. First, let's add a SetCameraTransform call in the Update function.

```
    void Update()
    {
        // If target is null, return from here: NullReference check
        if(!target)
            return;

        SetCameraTransform();

        // Handle mouse inputs for rotations
        if (useControlRotation && Application.isPlaying)
            Rotate();

        // Follow the target applying targetOffset
        transform.position = Vector3.SmoothDamp(transform.position,
            target.position + targetOffset, ref moveVelocity, movementSmoothTime);
    }
```

We will need some new variables in the follow settings region:

```
    [SerializeField] private float targetArmLength = 3f;
    [SerializeField] private Vector3 cameraOffset = new Vector3(0.5f, 0, -0.3f);

    private Vector3 endPoint;
    private Vector3 cameraPosition;
```

The SetCameraTransform function will for now compute the end position of the camera (which is a child of this object), relatively to the arm parameters.

```
private void SetCameraTransform() {
        // Cache transform as it is used quite often
        Transform trans = transform;

        // Offset a point in z direction of targetArmLength by camera offset
        // and translating it into world space.
        Vector3 targetArmOffset = cameraOffset - new Vector3(0, 0, targetArmLength);
        endPoint = trans.position + (trans.rotation * targetArmOffset);

        // Set cameraPosition value as endPoint
        cameraPosition = endPoint;

        // Iterate through all children and set their position as cameraPosition,
        // using SmoothDamp to smoothly translate the vectors.
        Vector3 cameraVelocity = Vector3.zero;
        foreach (Transform child in trans)
        {
            child.position = Vector3.SmoothDamp(child.position,
                cameraPosition, ref cameraVelocity, 0.2f);
        }
    }
```

Now we have a parameterizable arm for the camera. We will add collision management for the camera to avoid entering into walls.

Setting an arm spring

We will send raycast from the camera position to check and store the raycast and hit result in arrays:

```
    #region Collisions

    [Space]
    [Header("Collision Settings \n-----------------------")]
    [Space]

    [SerializeField] private bool doCollisionTest = true;
    [Range(2, 20)] [SerializeField] private int collisionTestResolution = 4;
    [SerializeField] private float collisionProbeSize = 0.3f;
    [SerializeField] private float collisionSmoothTime = 0.05f;
    [SerializeField] private LayerMask collisionLayerMask = ~0;

    private RaycastHit[] hits;
    private Vector3[] raycastPositions;

    #endregion
```

The two arrays have to be initilized at game start or when unity validates component's variables changes:

```
    void Start()
    {
        raycastPositions = new Vector3[collisionTestResolution];
        hits = new RaycastHit[collisionTestResolution];
    }

    private void OnValidate()
    {
        raycastPositions = new Vector3[collisionTestResolution];
        hits = new RaycastHit[collisionTestResolution];
    }
```

Before the SetTransformCamera call, we will compute raycast collisions:

```
    void Update()
    {
        // If target is null, return from here: NullReference check
        if(!target)
            return;

        // Collision check
        if (doCollisionTest)
            CheckCollisions();
        SetCameraTransform();
...
```

With:

```
    /// <summary>
    /// Checks for collisions and fill the raycastPositions and hits array
    /// </summary>
    private void CheckCollisions()
    {
        // Cache transform as it is used quite often
        Transform trans = transform;

        // iterate through raycastPositions and hits and set the corresponding data
        for (int i = 0, angle = 0; i < collisionTestResolution;
                i++, angle += 360 / collisionTestResolution)
        {
            // Calculate the local position of a point w.r.t angle
            Vector3 raycastLocalEndPoint = new Vector3(
                Mathf.Cos(angle * Mathf.Deg2Rad),
                Mathf.Sin(angle * Mathf.Deg2Rad),
                0) * collisionProbeSize;
            // Convert it to world space by offsetting it by origin: endPoint,
            // and push in the array
```

```
            raycastPositions[i] = endPoint + (trans.rotation * raycastLocalEndPoint);
            // Sets the hit struct if collision is detected between
            // this gameobject's position and calculated raycastPosition
            Physics.Linecast(trans.position, raycastPositions[i],
                            out hits[i], collisionLayerMask);
        }
    }
```

We will then update SetCameraTransform to take into account the camera collisions:

```
    private void SetCameraTransform()
    {
        // Cache transform as it is used quite often
        Transform trans = transform;

        // Offset a point in z direction of targetArmLength
        // by camera offset and translating it into world space.
        Vector3 targetArmOffset = cameraOffset - new Vector3(0, 0, targetArmLength);
        endPoint = trans.position + (trans.rotation * targetArmOffset);

        // If collisionTest is enabled
        if (doCollisionTest)
        {
            // Finds the minDistance
            float minDistance = targetArmLength;
            foreach (RaycastHit hit in hits)
            {
                if (!hit.collider)
                    continue;

                float distance = Vector3.Distance(hit.point, trans.position);
                if (minDistance > distance)
                {
                    minDistance = distance;
                }
            }

            // Calculate the direction of children movement
            Vector3 dir = (endPoint - trans.position).normalized;
            // Get vector for movement
            Vector3 armOffset = dir * (targetArmLength - minDistance);
            // Offset it by endPoint and set the cameraPositionValue
            cameraPosition = endPoint - armOffset;
        }
        // If collision is disabled
        else
        {
            // Set cameraPosition value as endPoint
            cameraPosition = endPoint;
        }

        // Iterate through all children and set their position as cameraPosition,
        // using SmoothDamp to smoothly translate the vectors.
        Vector3 cameraVelocity = Vector3.zero;
        foreach (Transform child in trans)
        {
            child.position = Vector3.SmoothDamp(child.position,
                cameraPosition, ref cameraVelocity, collisionSmoothTime);
        }
    }
```

If you try the code now, the camera will collide everything. You have to set the collision Layer of environment to Wall, the NPC to NPC and the player to Character, then set the script to collide with layer Wall and NPC.

You now have a camera that automatically reposition in function of obstacles! You could try setting its parameters when entering a building. Create setters functions to achieve that.

Debug features

In order to debug more easily our camera, we can draw the rays and a sphere around the camera.

We need some variables:

```
#region Debug

[Space]
[Header("Debugging \n--------------")]
[Space]

[SerializeField] private bool visualDebugging = true;
[SerializeField] private Color springArmColor = new Color(0.75f, 0.2f, 0.2f, 0.75f);
[Range(1f, 10f)] [SerializeField] private float springArmLineWidth = 6f;
[SerializeField] private bool showRaycasts;
[SerializeField] private bool showCollisionProbe;

private readonly Color collisionProbeColor = new Color(0.2f, 0.75f, 0.2f, 0.15f);

#endregion
```

Then we can use OnDrawGizmosSelected to draw the debug geometry when the SprimgArm object is selected.

```
private void OnDrawGizmosSelected()
{
    if(!visualDebugging)
        return;

    // Draw main LineTrace or LineTraces of RaycastPositions, useful for debugging
    Handles.color = springArmColor;
    if(showRaycasts)
    {
        foreach (Vector3 raycastPosition in raycastPositions)
        {
            Handles.DrawAAPolyLine(springArmLineWidth, 2,
                                   transform.position, raycastPosition);
        }
    }
    else
    {
        Handles.DrawAAPolyLine(springArmLineWidth, 2,
                               transform.position, endPoint);
    }

    // Draw collisionProbe, useful for debugging
    Handles.color = collisionProbeColor;
    if(showCollisionProbe)
    {
        Handles.SphereHandleCap(0, cameraPosition, Quaternion.identity,
                               2 * collisionProbeSize, EventType.Repaint);
    }
}
```

A dead zone for the camera to follow

If we move our character slightly, the camera will, however smoothly, catchup. This is not a behaviour we want. The player should be able to move a little without the camera moving.

We will implement a spherical deadzone around the character. If it moves while staying in this zone, the camera should not move.

Deadzone, a simplistic implementation

We can start a prototype of our new feature by mesuring the distance between the player and the spring arm, and forbiding the camera to move is this distance is inferior to a certain threshold.

We need a new variable in the Follow Setting region.

```
    [SerializeField] private float deadZoneSize = 2.0f;
```

Then we can do our distance check at the end of the update fonction, juste before the place we move the spring arm.

```
    void Update()
    {
        ...
        float distanceToTarget =
            Vector3.Distance(transform.position, target.position + targetOffset);
        if (distanceToTarget > deadZoneSize)
        {
            targetPosition = target.position + targetOffset;
        }
        else
        {
            targetPosition = transform.position;
        }
        // Follow the target applying targetOffset
        transform.position = Vector3.SmoothDamp(transform.position,
            targetPosition, ref moveVelocity, movementSmoothTime);
```

This keep the spring arm immobile until the character get out of the deadzone, but then when it moves to catch up to the character, it stalls at the frontier of the deadzone. We need to improve this behaviour for the spring arm to return to player position after is enters back in the deadzone.

Deadzone, a better implementation

We will consider the status of the camera regarding the deadzone as a state machine. We use an enum to figure the states of the state machine.

```
enum DeadZoneStatus
{
    In, Out, CatchingUp
}
```

We consider 3 states:

- The Out state is when the camera is outside the deadzone. In that case the targetPosition of the camera is the character.
- The CatchingUp state is when the spring arm has entered back the deadzone but must catch up the player position. It switched to In state when the spring arm reach a small zone around the player, defined by a new targetZoneSize variable (see below).
- The In state is when the spring arm has reached again the proximity of the player (using targetZoneSize) and stays in the deadzone.

We will also need new variables in the Follow settings:

```
    [SerializeField] private float deadZoneSize = 2.0f;
    [SerializeField] private float targetZoneSize = 0.1f;
    private DeadZoneStatus deadZoneStatus = DeadZoneStatus.In;
```

Now, let's implement this behaviour, still at the end of the Update function.

```
...
    // This value must be modified. If not, it is a handy debug.
    Vector3 targetPosition = Vector3.zero;
    // Compute distance
    float distanceToTarget = Vector3.Distance(transform.position,
                                    target.position + targetOffset);
    if (distanceToTarget > deadZoneSize)
    {
        deadZoneStatus = DeadZoneStatus.Out;
        targetPosition = target.position + targetOffset;
    }
    else
    {
        switch (deadZoneStatus)
        {
            case DeadZoneStatus.In:
                // In the deadzone, the spring arm stays still
                targetPosition = transform.position;
                break;
            case DeadZoneStatus.Out:
                targetPosition = target.position + targetOffset;
                // We are within deadzone, so switch to CatchingUp state
                deadZoneStatus = DeadZoneStatus.CatchingUp;
                break;
            case DeadZoneStatus.CatchingUp:
                targetPosition = target.position + targetOffset;
                // Switch to In state when near target
                if (distanceToTarget <= targetZoneSize)
                {
                    deadZoneStatus = DeadZoneStatus.In;
                }
                break;
        }
    }

    // Follow the target applying targetOffset
    transform.position = Vector3.SmoothDamp(transform.position,
        targetPosition, ref moveVelocity, movementSmoothTime);
```

Now the camera should catchup the character after it goes out of the deadzone.

A cinematic camera

We now want to create an external still camera, that will follow the player.

We create a new enum to be able to select the type of camera we want. We will anticipate the fact we will want a first person camera later.

```
enum CameraStatus
{
    ThirdPerson, FirstPerson, Camera1
}
```

We will need new variables :

```
    #region Camera Transition

    [Space]
    [Header("Camera Transition \n--------------")]
    [Space]

    [SerializeField] private Transform camera1;
    private CameraStatus cameraStatus = CameraStatus.ThirdPerson;

    #endregion
```

Now create a new game object in the scene and name it Camera1. Position it were you want the camera to be in the scene. Assign this camera to the Transform we have created in the SpringArm script.

We want to switch camera when the 1 / 2 keys are pressed. Pressing those keys will change the cameraStatus. Then, we will change the Update function in function of the status chosen. Once again, we use the targetPosition to move the camera.

```
void Update()
    {
        // If target is null, return from here: NullReference check
        if(!target)
            return;

        // Set targetPosition to a debug value
        Vector3 targetPosition = Vector3.zero;

        if (Input.GetKey(KeyCode.Alpha1))
        {
            cameraStatus = CameraStatus.ThirdPerson;
        }
        else if (Input.GetKey(KeyCode.Alpha2))
        {
            cameraStatus = CameraStatus.Camera1;
        }

        // Change camera in function of status
        if (cameraStatus == CameraStatus.Camera1)
        {
            targetPosition = camera1.position;
            transform.LookAt(target);
        }
        else if (cameraStatus == CameraStatus.ThirdPerson)
        {
            // Collision check
            if (doCollisionTest)
                CheckCollisions();
            SetCameraTransform();

            // Handle mouse inputs for rotations
            if (useControlRotation && Application.isPlaying)
                Rotate();

            float distanceToTarget = Vector3.Distance(transform.position,
                                                      target.position + targetOffset);
            if (distanceToTarget > deadZoneSize)
            {
                deadZoneStatus = DeadZoneStatus.Out;
                targetPosition = target.position + targetOffset;
            }
            else
            {
                switch (deadZoneStatus)
                {
                    case DeadZoneStatus.In:
                        targetPosition = transform.position;
                        break;
                    case DeadZoneStatus.Out:
                        targetPosition = target.position + targetOffset;
                        deadZoneStatus = DeadZoneStatus.CatchingUp;
                        break;
                    case DeadZoneStatus.CatchingUp:
                        targetPosition = target.position + targetOffset;
                        if (distanceToTarget <= targetZoneSize)
                        {
                            deadZoneStatus = DeadZoneStatus.In;
                        }
                        break;
                }
```

```
            }
        }

        // Follow the target applying targetOffset
        transform.position = Vector3.SmoothDamp(transform.position,
            targetPosition, ref moveVelocity, movementSmoothTime);
    }
```

We use the LookAt function for the cinematic camera to follow the player.

We could refactor this code to better manage camera states:

```
    void Update()
    {
        // If target is null, return from here: NullReference check
        if(!target)
            return;

        Vector3 targetPosition = Vector3.zero;

        if (Input.GetKey(KeyCode.Alpha1))
        {
            cameraStatus = CameraStatus.ThirdPerson;
        }
        else if (Input.GetKey(KeyCode.Alpha2))
        {
            cameraStatus = CameraStatus.Camera1;
        }

        switch (cameraStatus)
        {
            case CameraStatus.Camera1:
                targetPosition = UpdateCamera1();
                break;

            case CameraStatus.ThirdPerson:
                targetPosition = UpdateThirdPerson();
                break;
        }

        // Follow the target applying targetOffset
        transform.position = Vector3.SmoothDamp(transform.position,
            targetPosition, ref moveVelocity, movementSmoothTime);
    }

    Vector3 UpdateThirdPerson()
    {
        Vector3 targetPosition = Vector3.zero;

        // Collision check
        if (doCollisionTest)
            CheckCollisions();
        SetCameraTransform();

        // Handle mouse inputs for rotations
        if (useControlRotation && Application.isPlaying)
            Rotate();


        float distanceToTarget = Vector3.Distance(transform.position,
                                    target.position + targetOffset);
        if (distanceToTarget > deadZoneSize)
        {
            deadZoneStatus = DeadZoneStatus.Out;
            targetPosition = target.position + targetOffset;
        }
        else
        {
            switch (deadZoneStatus)
```

```
        {
            case DeadZoneStatus.In:
                targetPosition = transform.position;
                break;
            case DeadZoneStatus.Out:
                targetPosition = target.position + targetOffset;
                deadZoneStatus = DeadZoneStatus.CatchingUp;
                break;
            case DeadZoneStatus.CatchingUp:
                targetPosition = target.position + targetOffset;
                if (distanceToTarget <= targetZoneSize)
                {
                    deadZoneStatus = DeadZoneStatus.In;
                }
                break;
        }
    }

    return targetPosition;
}

Vector3 UpdateCamera1()
{
    transform.LookAt(target);
    return camera1.position;
}
```

First person camera

Actually, switching to a first person camera with our system is quite easy. If you set the arm length, the camera offset and the movementSmoothTime to zero, you will get a first person camera. Now you have to update the controls of your character in this mode!