Einführung in die Informatik für Games Engineering

Tutorials

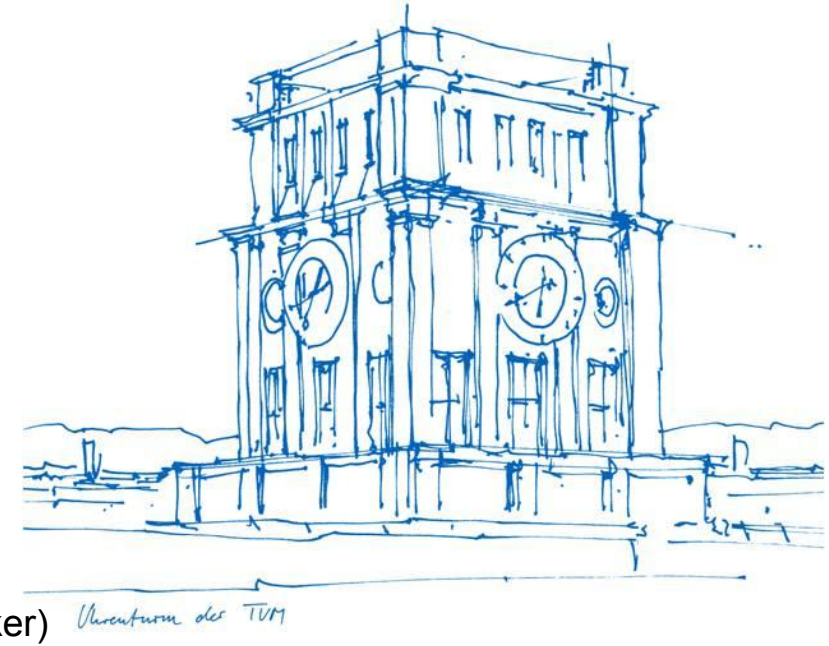
Gudrun Klinker (klinker@in.tum.de

Sven Liedtke (sven.liedtke@cit.tum.de)

Technical University of Munich

School of Computation, Information and Technology

Associate Professorship of Augmented Reality (Prof. Klinker)



Associate Professorship of Augmented Reality
School of Computation, Information and Technology
Technical University of Munich

What do you remember from last week?

What do you remember from last week?

- Debug breakpoints
- CCD & collision matrix
- UI
- Scene loading & build project

Lessons Learned – Issues

- `TMP_Text` can also be used instead of `TextMeshProUGUI` to reference TextMeshPro Text
- Don’t forget to add `using TMPro;` if not done automatically by your IDE

- `SceneManager.LoadScene(…)` accepts both build index and scene name as parameter
- Don’t forget to add `using UnityEngine.SceneManagement;` if not done automatically by your IDE

- Static variables need to be reset manually

Additional Task (fre k t during tutorial or at home)

#1 Show your score and live UI

#2 Show your menus

Questions?

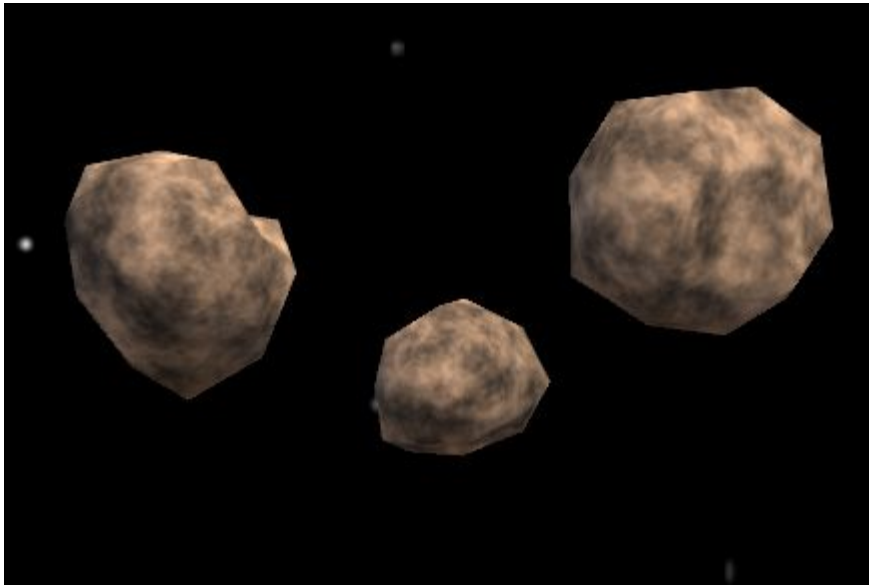
Today’s topics

- More asteroid variations
- Coroutines
- Enums and Player States

More asteroid variations

Tasks #1

1) Randomize rotation
2) Randomize scale
3) Increase speed on respawn



Randomize rotation

Note: If you've used the enemy script from the example project in Moodle, your asteroids should already rotate. Now it's time to randomize it.

Almost like how we randomized the **`speed`** with some differences:

1) Need only **`float maxRotationSpeed`** as parameter as the min is just the negative value
2) Store a global variable **`Vector3 rotationSpeed`** for the rotation
3) Use **`Random.Range`** on all three axes of rotationSpeed. The axes of a Vector3 can easily be edited e.g. **`rotationSpeed.x = value`**;

Then call **`transform.Rotate(...)`** in **`Update()`**. (Don't forget to make the speed frame independent!)

→ What do you see?

Randomize rotation – Solution example

1) Need only **`float maxRotationSpeed`** as parameter as the min is just the negative value

```
[SerializeField]
private float maxRotationSpeed;
```

2) Store a global variable **`Vector3 rotationSpeed`** for the rotation

```
private float speed;
private Vector3 rotationSpeed;
```

3) Use **`Random.Range`** on all three axes of rotationSpeed

```
rotationSpeed.x = Random.Range(-maxRotationSpeed, maxRotationSpeed);
rotationSpeed.y = Random.Range(-maxRotationSpeed, maxRotationSpeed);
rotationSpeed.z = Random.Range(-maxRotationSpeed, maxRotationSpeed);
```

Then call **`transform.Rotate(...)`** in **`Update()`**. (Don't forget to make the speed frame independent!)

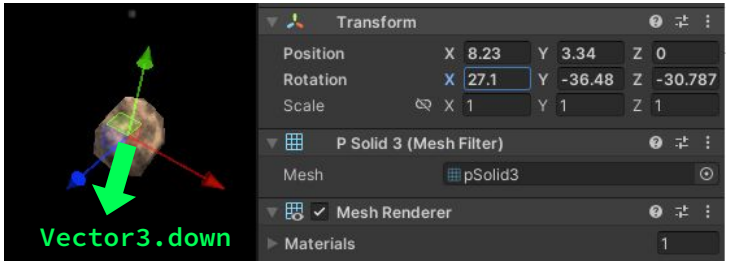
```
Vector3 amtToRotate = Time.deltaTime * rotationSpeed;
transform.Rotate(amtToRotate);
```

Associate Professorship of Augmented Reality
School of Computation, Information and Technology
Technical University of Munich

Randomize rotation

→ What do you see?

Asteroids are circling now instead of going down...



`transform.Translate(…)` uses local space if not specified.

Need to add `Space.World` in your movements so your enemy moves down independently from its current rotation/local coordinate system.

```
// move
float amtToMove = Time.deltaTime * speed;
transform.Translate(Vector3.down * amtToMove, Space.World);
```

Randomize scaling

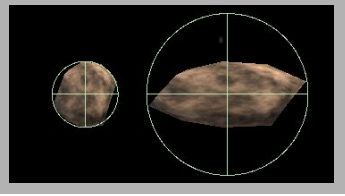
We set the object scale once every time it spawns.

1) Define a **`Vector2 minMaxScale`** parameter to your script.
2) Get a random value between min and max.
3) Multiply this value with **`Vector3.one`** and assign it to **`transform.localScale`**.

```
float scale = Random.Range(minMaxScale.x, minMaxScale.y);
transform.localScale = Vector3.one * scale;
```

Info: It's very difficult to get and set the world scale of an object. This is why we only use localScale. More info:https://docs.unity3d.com/ScriptReference/Transform-lossyScale.html

Info: We scale equally in all axes because we're using a SphereCollider. This collider remains round no matter how the object is stretched. The reason is how a SphereCollider works in the physics system. (Also why it's super performant)



Increase speed

We want to increase the difficulty of our game over time. So everytime an asteroid gets hit by a projectile, it increases its speed range.

1) Go to your projectile script where it collides with an enemy.

2) Increase the enemy's **`minMaxSpeed`** right before **`SetSpeedAndPosition()`** is called.

```
collideWith.minMaxSpeed.x += .5f;
collideWith.minMaxSpeed.y += .5f;
```

Optional: Changing minMaxSpeed by the projectile requires the variable to be public. But what if we want to keep it private? How would you do that?

Coroutines

Coroutines

“A coroutine allows you to spread tasks across several frames. In Unity, a coroutine is a method that can pause execution and return control to Unity but then continue where it left off on the following frame.

In most situations, when you call a method, it runs to completion and then returns control to the calling method, plus any optional return values. This means that any action that takes place within a method must happen within a single frame update.

In situations where you would like to use a method call to contain a procedural animation or a sequence of events over time, you can use a coroutine.

However, it’s important to remember that coroutines aren’t threads. Synchronous operations that run within a coroutine still execute on the main thread.”

https://docs.unity3d.com/Manual/Coroutines.html

Coroutines

Example:

Frame 1

```
private IEnumerator Wait()
{
    Debug.Log("Coroutine has started.");
    yield return null;
    Debug.Log("Coroutine ends.");
}
```

All operations before **`yield`** are executed.

Frame 1

```
private IEnumerator Wait()
{
    Debug.Log("Coroutine has started.");
    yield return null;
    Debug.Log("Coroutine ends.");
}
```

Unity leaves this function call and executes all remaining scripts in the frame.

Frame 2

```
private IEnumerator Wait()
{
    Debug.Log("Coroutine has started.");
    yield return null;
    Debug.Log("Coroutine ends.");
}
```

Unity reenters the function and continues where it left.

Coroutines

- The `yield` statement is a special kind of return that ensures that the function will continue from the line after the yield statement.
- There are many classes that helps you define how long this function yields:

```
yield return new WaitForSeconds(1);
yield return new WaitForEndOfFrame();
yield return new WaitForFixedUpdate();
yield return new WaitUntil(() => a == b);
yield return new WaitWhile(() => a == b);
// and more
```

- If `null` is returned, the function yields until the next frame.

Coroutines

- Coroutines can’t be called like normal functions. They must be started via `StartCoroutine(…)`.
- They can also be force stopped before its end.

```
// start the coroutine "Wait()". The function must have the return type IEnumerator
StartCoroutine(Wait());

// to stop a coroutine, it must be stored
Coroutine coroutine = StartCoroutine(Wait());
StopCoroutine(coroutine);

// or stop all coroutines that has been started in this object
StopAllCoroutines();
```

Tasks #2

- Extend our ship-to-asteroid collision
- Animate ship destruction:
  1) Instantiate explosion and remove the ship
  2) Respawn the ship in the middle after a few seconds if life is not zero

- Hint: To “remove” the ship we need to disable its renderer:

```
GetComponent<Renderer>().enabled = false;
```

Solution example

```
private IEnumerator DestroyShip()
{
    // ship explosion
    Instantiate(explosionPrefab, transform.position, transform.rotation);
    GetComponent<Renderer>().enabled = false;

    // lose life and wait for respawn
    lives--;
    yield return new WaitForSeconds(respawnTime);

    // check lives
    if (lives <= 0)
    {
        // game over
        SceneManager.LoadScene("Lose");
    }
    else
    {
        // respawn
        // reset position
        transform.position = Vector3.zero;
        GetComponent<Renderer>().enabled = true;
    }
}
```

```
private void OnTriggerEnter(Collider other)
{
    Enemy collideWith = other.GetComponent<Enemy>();
    if(collideWith != null)
    {
        // asteroid explosion
        Instantiate(explosionPrefab, transform.position, other.transform.rotation);
        // reset asteroid
        collideWith.SetSpeedAndPosition();

        // destroy ship
        StartCoroutine(DestroyShip());
    }
}
```

Note:
- Be cautious when copy&pasting! (naming, missing variables,...)

Issues

- Ship can still be destroyed even if it’s gone

  → Track the current player state and make its behaviour dependent on that state

  → We use enums!

Enums

Enums

- A set of custom named values, e.g.:

```
public enum State { Playing, Explosion, Invincible }
```

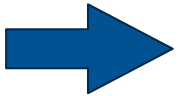
- Each value corresponds to an integer number which starts from 0 if not specified (here: Playing = 0, Explosion = 1, Invincible = 2)

```
// both possible
int playingStateNumber = (int)State.Playing;
State playingState = (State)playingStateNumber;
// not possible!
State undefinedState = (State)3;
```

Enums - Advantages

- No need to remember what an integer value stands for → readability
- No undefined values

```
// 0 = Playing
// 1 = Explosion
// 2 = Invincible
private int playerState;
```



```
public enum State
{
    Playing,
    Explosion,
    Invincible
}
private State playerState;
```

Tasks #3

- Track the current player state and make its behaviour dependent on that state
  1) Player has a state that can be **`Playing`**, **`Explosion`**, or **`Invincible`** (check previous slides)
  2) **`Playing`** state: default, player can be hit
  3) **`Explosion`** state: player can't move, shoot or be hit
  4) **`Invincible`** state: player can't be hit by asteroids

Playing state

1) Set the state to **`Playing`** right at the beginning (usually done automatically because **`Playing`** is the first/default state value)

2) Also set it to **`Playing`** at the end of the respawn coroutine

3) Make the player only collide with asteroids when the state is **`Playing`**

```
Enemy collideWith = other.GetComponent<Enemy>();
if (collideWith != null && playerState == State.Playing)
{
```

Explosion state

1) Change the state to **`Explosion`** in the beginning of the DestroyShip() coroutine
2) Only execute everything you had in **`Update()`** if the state is not **`Explosion`**.

```
if (playerState != State.Explosion)
{
    // ...
}
```

Invincible state

When the player respawns, there is a short time where the player can move and shoot again but not be hit by asteroids.

1) Extend your respawn coroutine so that it waits another couple of seconds after enabling the renderer
2) Change the state to `Invincible` between the two `yield`s

Additional Tasks (free work time during tutorial or at home)

- Add a “missing” counter that tracks how many asteroids has passed through without colliding and display it in the UI
- Polish the destroy&respawn animation:
  - Let the ship respawn below the view and automatically move it up into the screen.
  - Hint: how to move up over time until a start position

```
while (transform.position.y < 0)
{
    float amtToMoveY = Time.deltaTime * speed;
    transform.position += Vector3.up * amtToMoveY;
    yield return null;
}
```

- Add a blinking animation that indicates the ship’s invisibility
  - Hint: Start a separate coroutine right before moving up that blinks until state is `Playing`