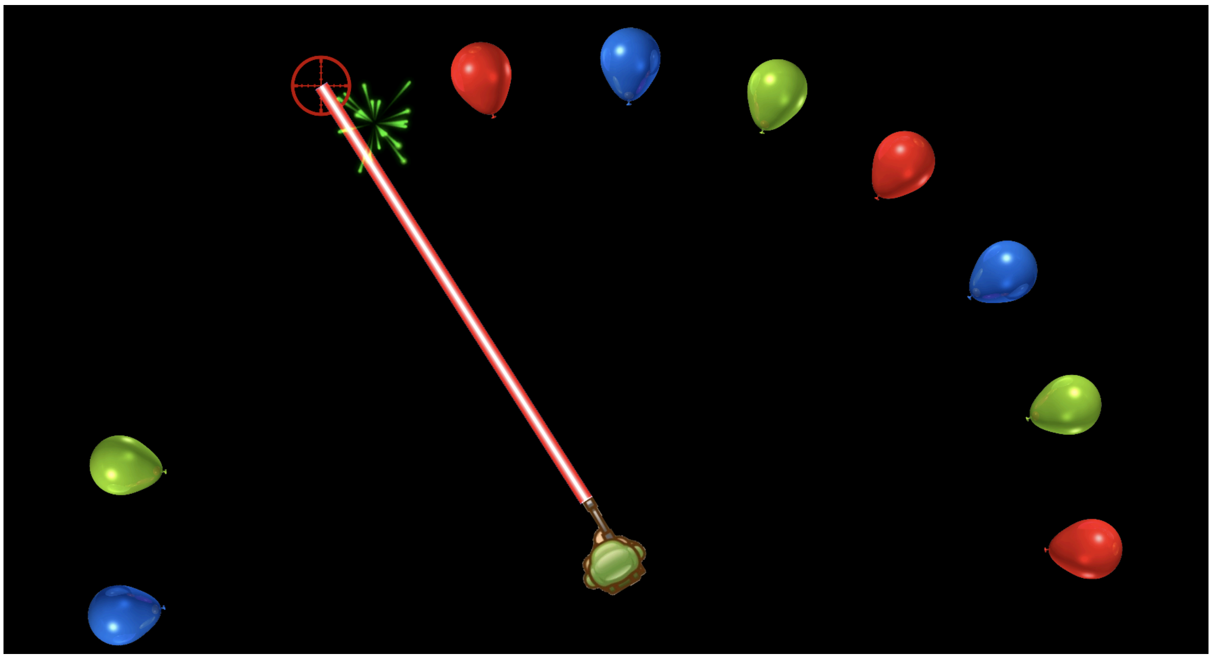
Laser Balloons

In this project you will familiarize yourself with Vector math, rotation and Raycasts.
The goal is to let the soldier shoot all the balloons with his laser gun.



The project is already set up partly:

- The `Soldier` script controls a `LineRenderer` that is used to draw a line from the guns end to the target in the `UpdateCrosshair()` method.
- The Balloons come in three colors. They have a `Balloon` script attached that has a method `public void Pop()`. This will destroy the Balloon and spawn a hit effect.
- The Balloon hit effect will play a simple sprite animation when spawned.
- There is a GameManager to restart the scene using Space, Return or Esc.

Basic Exercise

Setting up

1. Create a new 2D project and import the Unity Package provided for the lab.
2. **Open the MainScene in the Scenes folder**, here you can find a basic setup for the exercises.
3. Explore the setup, what objects are in place, what are prefabs? Who has which script attached?

1. Moving the crosshair with the mouse

First let us set up movement for the crosshair in the `Soldier` script. The function `UpdateCrosshair(Vector3 newCrosshairPosition)` in `Soldier` can place the Crosshair (`crosshair` GameObject) and update the laser line when given the proper

position. `GetMouseWorldPos()` in the script can give us that position. Make use of the functions in `Update()` to move the crosshair.
When set up correctly, you will be able to move the Crosshair graphic when moving the mouse in the running game.

2. Popping Balloons (using raycasting)

Make the balloons pop! In `Update()` of `Soldier` use a 2D RayCast to check if any balloons got hit by the laser.
Preparation and tips:

- Check out https://docs.unity3d.com/ScriptReference/Physics2D.Raycast.html
- Check out the lecture slides for details on how to use the function.
- `laserStart` and `crosshair` are of type `GameObject`. To access their position use `lasterStart.transform.position` and `crosshair.transform.position`.

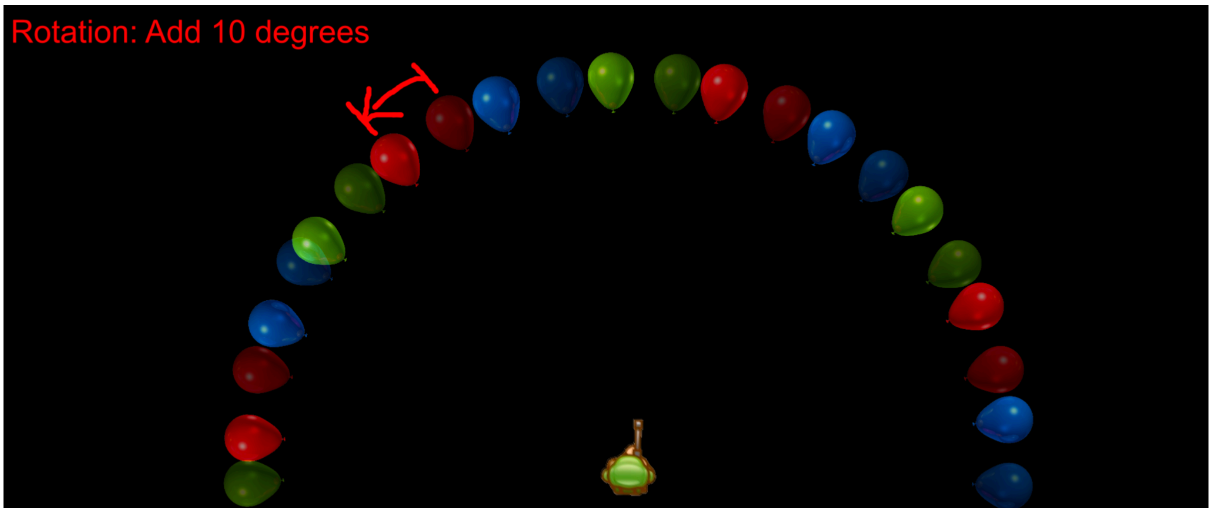
The Raycast:
To make the setup for a raycast easier to read, let us set up the variables:
`origin`: The start of the raycast at `lasterStart.transform.position.`
`target`: The target of the raycast at `crosshair.transform.position.`
`direction`: The direction of the raycast from origin to target.
`distance`: The distance between origin and target.

- The Raycast should start at `origin` and go into the `direction` to `target` for the `distance`.
- The `Physycs2D.Raycast` will return an object of type `RacastHit2D` usually called `RacastHit2D hit`. This contains a field `collider`, that stores the Collider the ray has hit (if any, else its `null`).
  - Create a variable `RacastHit2D hit` to store our Raycast result in.
  - Check if `hit.collider == null`. If that is NOT the case, we hit a Balloon.
  - Call the `Pop()` function on the attached `Balloon` script. You can use `hit.collider.gameObject.GetComponent<Balloon>()` to grab a reference to the script from the object you hit.
- A regular Raycast will stop at the first Collider it hits. The raycast might hit the soldier if you put a collider on it. You can either make sure there are no colliders on the soldier or use a `LayerMask`:
- (*optional*): You can create a Layer *Balloon* to put the balloons on and then add a `LayerMask` to the `Raycast` to only test against this Layer, to make sure the `Raycast` will only hit balloon objects. (Read more on this in the appendix).

3. Balloon Merry Go Round (using quaternions)

Let the balloons rotate around the soldier. As all the individual balloons are children of the ***Balloons*** object in the scene, they will follow if we rotate that object. To handle the rotation, there should be a `BalloonCenter` script on ***Balloons***. Modify this script using Quaternions according to the following instructions.

For rotations in Unity you will have to use a `Quaternion`. The function `Quaternion.AngleAxis(float angle, Vector3 axis)` will create a rotation (`Quaternion`) that describes a rotation of `angle` degrees around that `axis`. In 2D we always want to rotate around the z axis, which equals to `Vector3.forward`. So we usually use:
`Quaternion.AngleAxis(angle, Vector3.forward)`

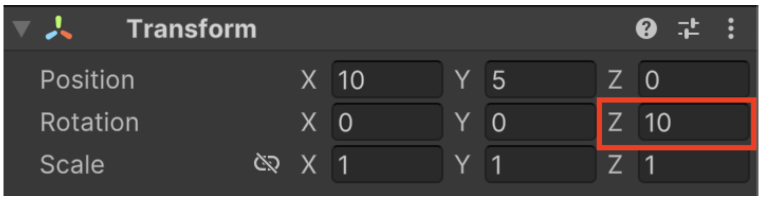


Positive angles rotate counter-clockwise!

We can easily create a Quaternion that describes a rotation by 10 degrees around Z:
`Quaternion q = Quaternion.AngleAxis(10, Vector3.forward);`

We can assign a rotation to a transform by assigning this quaternion:
`transform.rotation = Quaternion.AngleAxis(10, Vector3.forward);`

Assigning a fixed rotation will only rotate our object once. Then it is fixed on that one rotation of 10 degrees:



Transform

Position	X	10	Y	5	Z	0
Rotation	X	0	Y	0	Z	10
Scale	X	1	Y	1	Z	1

We want to <u>continually</u> rotate the balloons. So we have to <u>increment</u> the rotation. This works the same as incrementing the position (compare Flappy Bird lab):
`transform.position += direction * speed * Time.deltaTime;`
Here we increment the `position` every frame by a fraction of `speed` into `direction` based on fps (`deltaTime`).

Rotations are stored in Quaternions, not Vector3s or Vector2s (as `direction`). To add a rotation to an object, we need to respect how Quaternions operate:
To add a quaternion, we need to multiply them!

```
transform.rotation *= Quaternion.AngleAxis(angle,
Vector3.forward);
```

We also have to scale the rotation (`angle`) by `Time.deltaTime`, to ensure we respect different fps:

```
float angle = rotationSpeed * Time.deltaTime;
transform.rotation *= Quaternion.AngleAxis(angle,
Vector3.forward);
```

Working with quaternions

1. Quaternions can be multiplied to “add” their rotation on top of each other. A Quaternion that rotates 40 degrees around the z axis, multiplied with a Quaternion that rotates 20 degrees around the z axis, will result in a Quaternion that rotates 60 degrees around the z axis.
2. Quaternions multiplications is NOT commutative! (a*b != b*a, unless the rotation takes place on the same axis).
3. Quaternions can NOT be scaled like Vectors! (They don’t have the same kind of scalar multiplication). To scale a Quaternion you will have to scale the angle before you create it.

4. Rotating the soldier (more on rotations, angles and atan2)

Make the Soldier look into the direction of the crosshair.
This can be done two ways:
The easy one by just setting the transform direction directly.
The harder, but more flexible and mathematically correct way of using quaternions.

1. (Easy): Using setting of transform.up or transform.right

A transform can be relatively easily rotated by simply setting the value for its local right or up directions. You can apply your direction to `transform.up` (set local y axis to face given direction) to make the soldier look that way. This only works in Unity, more on that in the appendix.

2. (Hard, but better and more flexible): Using quaternions, angles and atan2

Better than setting the local axis (which in turn will make the object rotate to confirm with the new settings and still have a right angled cartesian coordinate system) is to give the object the new desired rotation.
For this you will have to use the function atan2 to calculate the correct angle from the direction vector, then apply it to the soldier using `Quaternion.AngleAxis(float angle, Vector3 axis)`.

1. **Angles from directions:** First, calculate the correct direction to look in, that is from the Soldier (`transform.position`) to the crosshair (`crosshair.transform.position`) and save that in a new local variable of type

`Vector3`. Normalize that vector (it has a `Normalize()` function). Now we need to convert this direction into an angle to apply it as a rotation to the soldier:

2. **Calculating the correct facing angle:** Since we are in 2D we can use the very helpful function `Mathf.Atan2(float y, float x)` (yes y is first). Atan2 is an extension of the tangents that allows conversion of a Vector2(x,y) into a signed angle, we can then use that angle to rotate the Soldier. *(more on atan2 in appendix).*
3. **Radians to Degrees**: The angle Atan2 gives us is in **radians**, the angle we need is in **degrees**, so multiply it with `Mathf.Rad2Deg` to convert.
4. **Applying the angle to the Soldier GameObject.** In order to apply the angle you will need to convert it into a Quaternion. The best suited for us is `Quaternion.AngleAxis(float angle, Vector3 axis)`. The axis we want is the world's global z-axis, this equals to `Vector3.forward`, or `new Vector3(0,0,1)`. Just assign the resulting Quaternion to `transform.rotation`.
5. **Testing**: Test your result, the soldier should rotate with the crosshair moving, but look to the left. If it spins rapidly or does rotate very slow you have an error in the math.
6. **Why does he look left?** Now we have to take the soldier sprite default facing direction into account. The resulting angle is the direction angle relative to the right facing direction or the x-axis. The soldier sprite faces up by default (check sprite), so we need to subtract `Mathf.PI * 0.5f` from the angle (in radians) or 90 degrees to align it with the sprite.

Bonus Goals

Only shoot when you are asked to

Make the soldier shoot the laser (laser graphics and raycast) only when you hold down the left mouse button (or any other button). You can use `Input.GetMouseButton(0)` to check if the left mouse button is held down. The laser `lineRenderer` can be turned off and on using `lineRenderer.enabled = true` / `lineRenderer.enabled = false`.

Poof, so pretty

A: Adjust the `Poof` script to play back a sprite animation from the respective poof sprite frames Instead of just destroying the poof object after 1 second. You can use the logic to cycle through sprites in the `Update` function from the lecture.

B: Instead of using the `Update` function to play the sprite animation, create a `Coroutine` to cycle through the frames and then destroy the GameObject afterwards at the end of the `Coroutine`.
Coroutines allow you to wait for a set amount of time in the middle of your code and to run timers and timed code outside of the `Update` function.
Find out how to use Coroutines here: https://docs.unity3d.com/Manual/Coroutines.html

Appendix

Tips for rotating objects in 2D:

In 2D space you always want to rotate around the global z-axis. So `Quaternion.AngleAxis(float angle, Vector3.forward)` is your friend all the way. Unity is an engine originally developed for 3D games. This means also the 2D part of the engine is actually using 3D for all calculations. Calculating rotations in 2D can be done with basic trigonometry, but because of the 3D Unity core we need to apply them using Quaternions, the best way to safely calculate 3D rotation and orientation.

Why setting transform.up/transform.right to rotate is a bad idea

Each object in a coordinate system like Unitys has its own local coordinates defining its position, its rotation and scale (this is called the “transform”). An object's local coordinate system and its up (y), right (x) and z (forward) axis depends on the objects rotation. If one of these axes is set directly using eg. `transform.up = new Vector3(1,0,0);` Unity will reorient the local coordinate system, meaning rotate the object. The engine functionality makes sure all locas axes are in a 90 degree angle and the transforms consistency is preserved. BUT in any other environment without these helper functions you will have to use a proper rotation ether by matrix or quaternion to rotate an object. Furthermore any more complex calculations for rotations, besides the simple setting of a direction, will require Quaternions.

Raycasts and LayerMasks

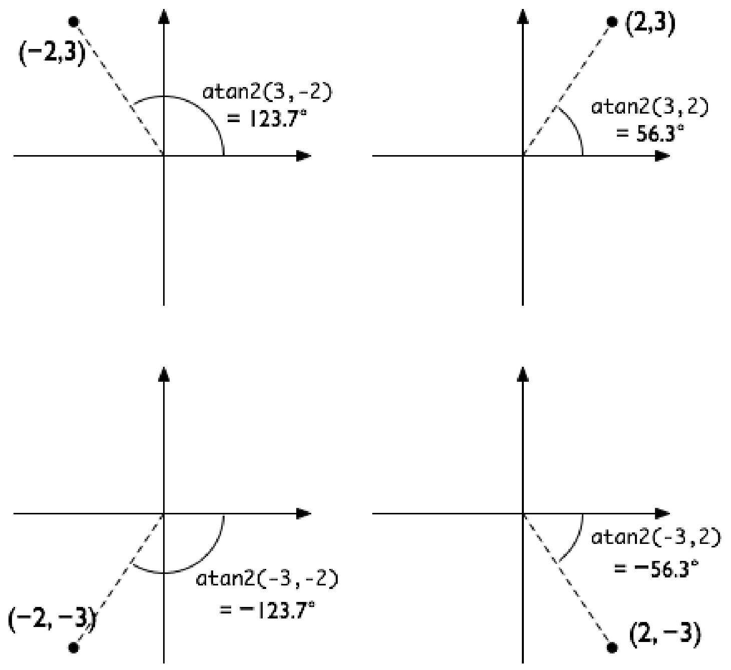
All physics interactions can be influenced by LayerMasks. This means objects can be sorted into layers and rules for which layer interacts with which can be applied. All physics based tests like the Raycast can use LayerMasks to only test against objects on certain layers. The respective testing mask is then given to the Raycast function as a parameter.

Definition of Atan2

Atan2 returns the signed angle of a vector V=Vector2(x,y) to the positive x-axis (to the right) **in radians**. Save the result of the atan2 in a new local variable of type `float`. It is often superior to the normal dot product since that can only deliver an unsigned angle and we would not know whether the rotation is clock- or counterclockwise.

It is defined as:

$$\operatorname{atan2}(y,x) = \begin{cases} \arctan(\frac{y}{x}) & \text{if } x > 0 \\ \arctan(\frac{y}{x}) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan(\frac{y}{x}) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ +\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases}$$


(−2,3)
atan2(3,-2)
= 123.7°
(2,3)
atan2(3,2)
= 56.3°
(−2, −3)
atan2(-3,-2)
= −123.7°
atan2(-3,2)
= −56.3°
(2, −3)