Quadrilateral Warfare - Documentation

Christoph Kirchberger 11705534
Nico Nößler 01431141

The quadrilaterals had enough of the nonsense the evil triangles have set up. Interior angles of only 180 degrees? What blasphemy. They need to be eliminated for good.

Geometry Birds is heavily inspired by two games; **Geometry Wars** and **Angry Birds**.
It combines the graphical style of Geometry Wars with the gameplay of Angry Birds.

The goal is to eliminate all triangular entities in the level to complete the game.

Controls

Mouse to **click** and **drag** Player Gameobject
Escape to pause/unpause the game

Optional Controls:
WASD to Move the Player Gameobject in **-X/X** and **-Y/Y** respectively
Q/E to rotate the Player Gameobject
MouseWheel to scale Player Gameobject

Screenshot of the initial game state, player square on the left, enemies on the right

Gameplay

The Goal is to launch your Quadrilateral (blue on the left side) into the evil triangles on the right side. To do so you need to tension your spring and launch the Quadrilateral into the house of the triangles like with a slingshot.

The house is built out of solid, indestructible blocks connected with springs (those can be broken when they reach too much tension!).

If you don’t hit or need a new Quadrilateral, just click on the used one and you’ll get a new one.

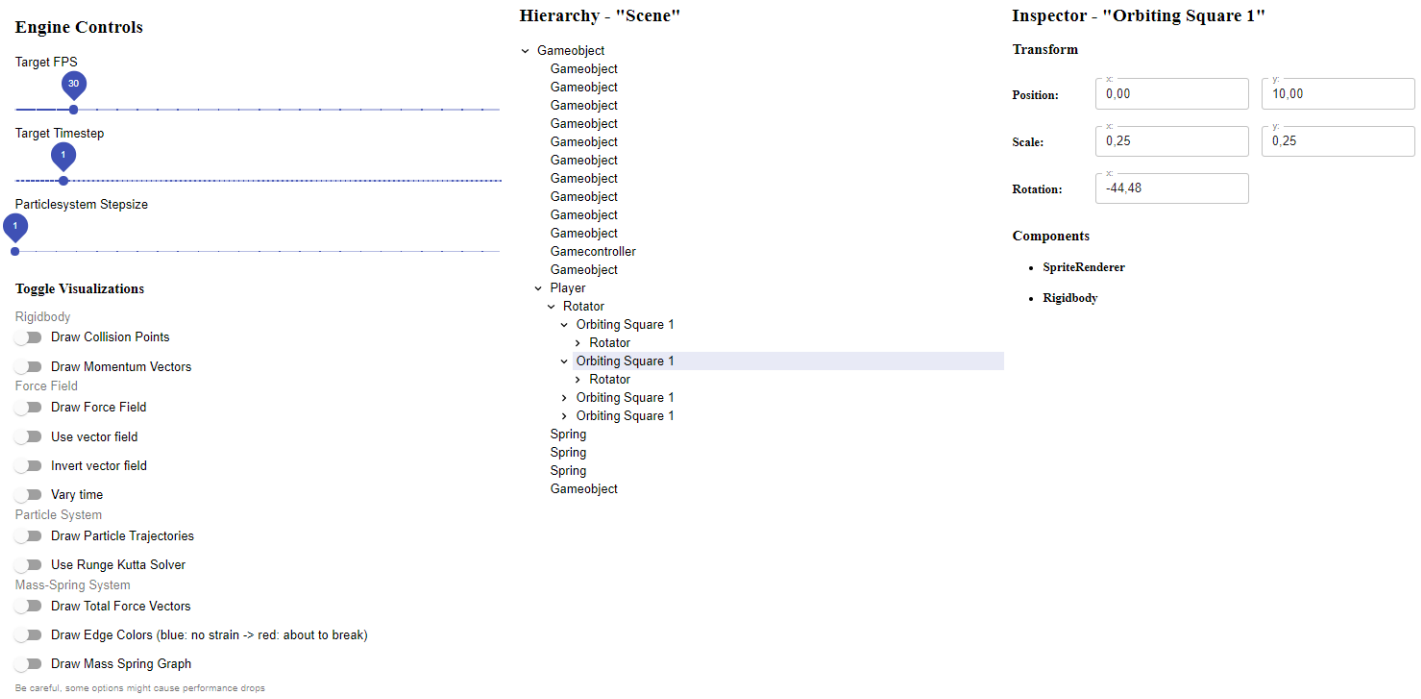
If you destroy all the Triangles in the Level you have won, congratulations!

Pull the Player Quadrilateral to the left to tension the spring!

Try to hit the squares and get the building to collapse! But be careful, as the collapsed blocks might be a better cover for those triangles!

Optional Customizations

If you’re into the tech stuff you can scroll down on the gamepage to reveal some fancy debug tools we included with this which were heavily inspired by the Unity game engine.

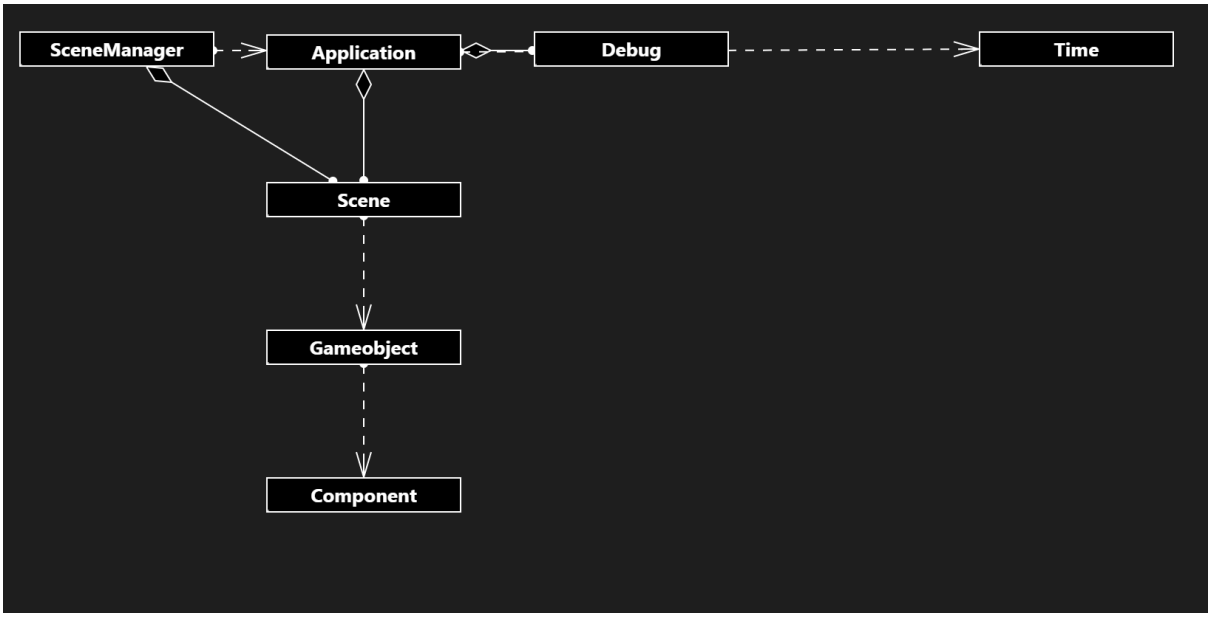


- Engine Controls
  - Set your Target FPS and Target Timestep for the physics engine and observe how it affects the game (and creates glitches, which are funny sometimes)
  - Toggle Visualizations of included engine features (be careful, some can lead to laggy rendering)
- Hierarchy View
  - See all the Gameobjects which are in the scene, and all their children and their children and so on. A click on one of these will activate the Inspector right to it.
- Inspector View
  - The Inspector shows the selected gameobject in the Hierarchy View
    - It shows the Transform (relative to its parent) and the Components which are added to this specific gameobject.

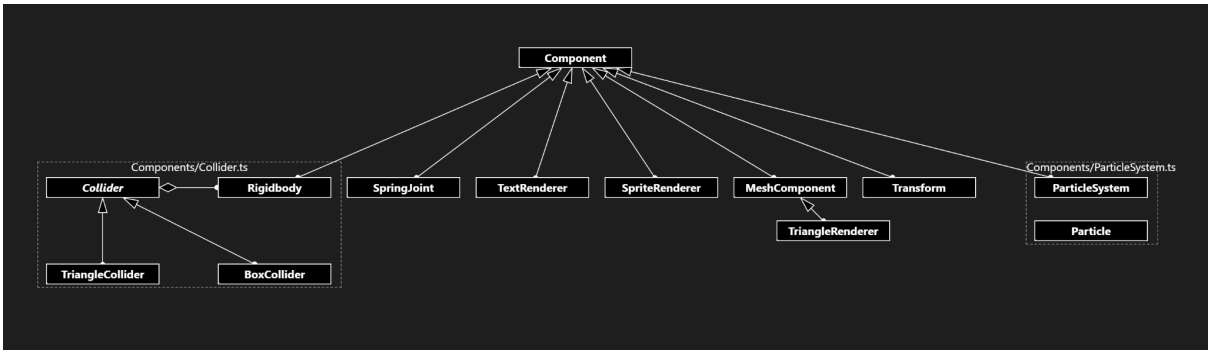
Tech-Documentation

General implementation notes

The whole engine is slightly oriented on Unity's structure. As can be seen in the class diagram shown below, it goes: Application -> hosts a Scene -> holds Gameobjects -> holds Components.
All time-relevant values are stored in the static Time class.
The Debug class is solely holding parameters for certain visualizations.



As can be seen below, every module derives from Component.ts, which makes it easy to add/remove modules to Gameobjects and handle animation or render updates.



Feature: Rigid-body dynamics

- ***src/Engine/Components/Rigidbody.ts***
- ***src/Engine/Components/Collider.ts***
- Linear-/Angular velocities calculated by Velocity-Verlet-Integration from force and torque at position, mass and inertia
- ***previously***: Euler Integration:
  - $v_{t+1} = v_t + F(p_t) / m * \Delta t$
  - $p_{t+1} = p_t + v_t * \Delta t$
- ***now***: Velocity-Verlet-Integration:
  - $p_{t+1} = p_t + v_t * \Delta t + 0.5\, a_t * \Delta t^2$
  - $a_{t+1} = F(p_{t+1}) / m$
  - $v_{t+1} = v_t + 0.5(a_t + a_{t+1}) * \Delta t$

- Collision detection
  - Separating Axis Theorem to retrieve MTV
  - Use MTV to translate bodies out of each other
  - Use MTV to calculate contact point & collision normal
  - From contact point & collision point, compute linear & angular velocities of colliding bodies

Feature: Physically-based particle dynamics

- ***src/Engine/Components/ParticleSystem.ts***
- ***src/Engine/ForceField.ts***
- Positions and Velocities for each particle calculated from positional forces, solved by Runge-Kutta (RK4) or Euler ODE
- Particle System class which creates & handles Particle instances, updates their lifetime, applies given “over-lifetime” functions (e.g. a changing color depending on the remaining lifetime) and updates their positions and velocities
- The Particle System can either loop or be triggered manually (e.g. on a collision)

Feature: Mass-spring systems

- ***src/Engine/Components/SpringJoint.ts***
- ***src/Engine/Components/Rigidbody.ts::GetLocalForce(pos)***
- Force calculated by Hooke's Law:
  - $F = -k * (\Delta x - d)$
  - k… Spring constant
  - Δx... actual distance between mass points
  - d… spring resting length
- A Rigidbody collects all the accumulated spring forces of its attached springs, adds global forces and applies them via Verlet-Integration

Feature: Hierarchical transformations

- ***src/Engine/Components/Gameobject.ts::UpdateTransform()***
- Main idea: Calculate "absolute Transform" from local Transform and parent's absolute Transform:
  - $absP = parentAbsP + (localP * parentScale).RotateBy(parentRotation)$
  - $absScale = parentScale * localScale$

Sources / Inspirations

https://gafferongames.com/post/integration_basics/
http://lampx.tugraz.at/~hadley/num/ch8/rk4ode2.php
https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods
https://www.geeksforgeeks.org/runge-kutta-4th-order-method-solve-differential-equation/
https://www.myphysicslab.com/engine2D/collision-en.html
https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects