DAT155 Graphic methods last updated 1.109

All exercises are curriculum

Exercise 5

Uke 41

Fint hvis dere alltid diskuter gruppen før dere sender spørsmål til oss om oppgavene
Deadline to submit: 14.10 Groups of mostly 4 members, but none more.
All members have to present for demonstration on Tuesday 13.10
Submission: one zip file: videos, projects (code) and a report (pdf)
A short report 2-3 pages with descriptions, explanation of your code and graphic techniques you have used and pictures of results from all three assignments.

- **Introduction to Threejs.**
- **Develop a game**
- **Study new topics and some advanced teqniques**

- **Ref: Lecture Intro Threejs**
  https://threejs.org/
  Download The Solar System from the Module Code

Assignment 1 (Demonstration and submission)

- Make some more planets in the Solar System orbiting the Sun, threejs.
-

  Option 1 Continue with the code you worked on in the threejs seminar.
  Option2 Download the start code from the Code module ,**threejs-solarsystem-filledIn.zip** on the **module Three.js "kurs"**

  Preferably use webgl 2.0.
  Try to make a custom shader.

Assignment 2 (Demonstration and submission)

You must complete task 2 and 6.

Introduction

In this assignment you will develop a game using WebGL2. A big part of the game has already been implemented. **Download from Canvas** (**module code).** You are to implement your solutions in this project.

The purpose of this assignment is to gain insights into how a graphics engine works, and how a game can be developed on top of this abstraction.

You can press 'F' in the game to toggle between playing and a free camera mode.

If you're interested in how you can write your own graphics engine take a look at the glTF2.0 specification. The specification contains lots of useful implementation notes.

Task 1: Health system (optional)

We can provide a function to a CollisionObject to execute if two objects intersect.

```
someCollisionObject.setOnIntersectListener((delta, entity) => {
  // 'entity' is the CollisionObject I'm intersecting with.
});
```

Implement a health tracking system for the player.

For example, the player can start at 3 hp (health point). Then upon impact with a block 1 hp is removed. Finally, when the player reaches 0 hp the game can either restart automatically or present a "game over" screen.

Optional: Add a period of invulnerability after losing a hit point.

Hints: The "game over" screen can be implemented with a simple div-element overlaying the canvas.

Task 2: Intersection between entities (obligatory, demonstration and submission)

The PhysicsManager is, amongst other things, responsible for checking if CollisionObject instances intersect. Instances of CollisionObject will henceforth be referred to as entities. In the current implementation we use a double for-loop to check if an entity intersects with other entities.

```
for (let i = 0; i < entities.length; i++) {
    for (let j = 0; j < entities.length; j++) {
        if (i === j) continue;
        if (intersect(entities[i].boundingBox, entities[j].boundingBox)) {
            ...
        }
    }
}
```

This gives us a run time of $O(n^2)$. The number of intersection checks will increase quadratically. How can we improve the performance?

Well, most of the entities in the scene are static, and so because they cannot move they will never cause a new intersection. A dynamic entity on the other hand can by moving cause an intersection to occur. In other words, we only need to check if the dynamic entities intersect with the other entities.

```
for (let i = 0; i < entities.length; i++) {
    if (entities[i].dynamic === false) continue; // skip static entities
    for (let j = 0; j < entities.length; j++) {
        if (i === j) continue;
        if (intersect(entities[i].boundingBox, entities[j].boundingBox)) {
            ...
        }
    }
}
```

In our case, since the player is the only dynamic entity, we've effectively reduced the number of checks to (n – 1) reducing the running time to O(n).

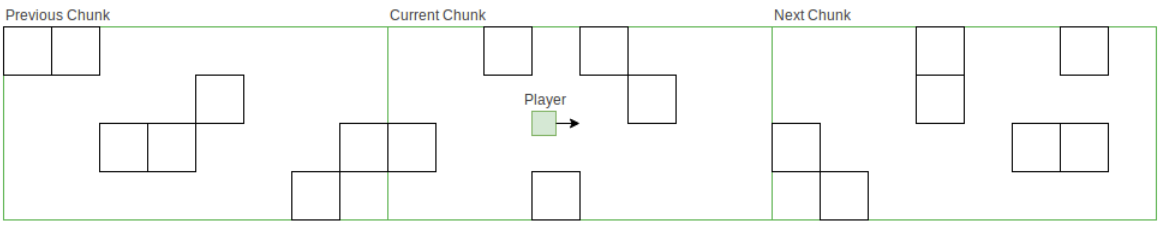
Implement the improved version.

Hint: One way to solve this is by adding a Boolean property called "dynamic" to CollisionObject, and then only doing intersection checks if the entity is dynamic.

Optional: Currently the intersection checks only support AABBs (axis-aligned bounding boxes). Investigate how we can add support for non-axis aligned bounding boxes as well.

Task 4: Obstacles (optional)

The ObstacleManager's task is to generate and manage obstacles in the game. It does so by generating "chunks". A chunk is a bunch of blocks spread out randomly over a x*y area with a certain density. To avoid performance issues only 3 chunks are active at a point in time.



As the player moves forward it will eventually move into the "Next Chunk", at this point "Previous Chunk" is removed, "Current Chunk" becomes the previous chunk, "Next Chunk" becomes the current chunk, and a new "Next Chunk" is generated.

This comes with a drawback though. As the player moves forward the new chunk will visibly pop into existence causing an unwanted visual stutter. You are to make it feel more seamless by modifying ObstacleManager so that it generates one row of blocks at a time. Additionally, investigate what happens when the speed of the player increases. Can this cause unwanted behavior?

Hint: You can keep track of the blocks by using a two-dimensional array where the outermost array acts like a queue. You add a new row in front, and pop off and destroy the last one.

Optional 1: Creating (sizeX * sizeY) * density Mesh-instances per chunk requires a lot of memory, and can cause stuttering due to the garbage collector kicking in. Try reusing Mesh-instances.

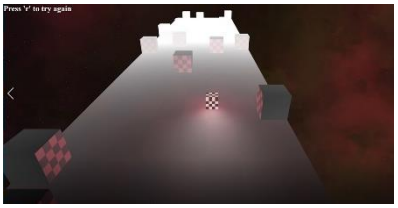
Optional 2: Currently the obstacles are generated randomly, this means that there is a possibility of the game being unbeatable at some point. I.e. it can be impossible to navigate around the obstacles. Reduce or remove the chance that the obstacles are unbeatable.

Task 5: Sphere primitive (optional)

The player is represented with a box. Replace the mesh representing the player with a sphere, and make it roll like a ball.

Task 6: WebGL techniques (obligatory, demonstration and submission)

Implement fog in the Phong-shader. The result should look something like this:



Although, you can adjust the fog-blending color to be closer to the background.

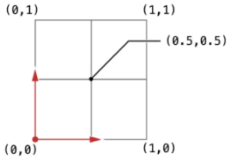
Optional: Experiment with different fog-functions (linear, logarithmic, …)

Assignment 3 (No Demonstration and submission)

Some advanced topics and techniques

Det anbefales at dere studerer disse prosjektene, diksuterer de og kan forklare de.
Noen har dere ikke har forutsetninger for å forstå ennå, men dere vender tilbake til de senere.
Koden finner dere på lærebokens side: http://interactivecomputergraphics.com/Code/

1. **Off screen rendering**.
   Hent ned kode fra lærebok fra mappen 07.renderv1v2, render3 og
   frameBufferObject (3d-objekt) http://rodger.global-linguist.com/webgl/ch10/FramebufferObject.html
   Koden hentes ned her: https://sites.google.com/site/webglbook/
2. **Environment mapping**
   reflectionMap
   reflectionMap2 (Kanskje litt upresist kommentert i README.txt)
   Interessant kode å studere her: http://math.hws.edu/graphicsbook/source/webgl/cube-camera.html
3. **Shadow mapping**
   shadowMap fra mappen 08
   Forklaring her http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/#basic-shadowmap
4. **Sprite**
   En sprite er et 2D-bildet (gjerne av en figur) som kan bevege seg på skjermen (eks. i 2D-spill som Pac-Man).
   En musemarkør er også en sprite.
   pointSprite9 fra mappen 10



5. **Projektiv tekstur**
   ProjectiveTexture fra mappen 07. Se forelesning F11 Tekstur1
   Kommentar. I vertex shader:
   Klippekordinater er i homogene koordianter [x,y,z,w] der x,y og z i interavllet [-w,w]
   Ved prespektiv projeksjon; [x/w,y/w,z/w] får vi normaliserte enhetskoordinater, ndc:
   Disse koordinatene er i i intervallet [-1,1] og webggl lager disse.
   I vertex-shader skal vi selv lage ndc-koordinater som skal brukes for å lage teksturkoordianter:
   Der står det bla.a:
   vProjTexCoord = 0.5 * objLightPosition.xyz + 0.5;
   Men vi forventet at det stod:
   vProjTexCoord = 0.5 * objLightPosition.xyz/objLightPosition.w + 0.5;
   Bakgrunnen er at gls-funksjonen utfører perspektivisk divisjon, så det slipper her.
6. **Instansiering**
   Forklaring her**:** https://learnopengl.com/Advanced-OpenGL/Instancing
   https://www.saschawillems.de/blog/2015/04/25/geometry-instancing-with-webgl-2/
   teapotInstance2 fra mappen 10
7. **Bump mapping**
   Se forelesning F17 som blir lagt ut.Hent ned prosjektet bumpMap i 07. Hent ned prosjektet
   PhongShadingWithNormalMap fra modulen Code. Studer shader-koden.

http://www.realtimerendering.com/blog/webgl-2-new-features/