PA199 – Advanced Game Development

Project Assignment

Semester: Autumn 2023

document revision: 1.0.1

The aim of the assignment is twofold:

- develop a core of a game engine in C++, including basic graphics and physics components, and
- develop a simple game using your own engine.

The engine should cover the general motion of rigid bodies in three-dimensional space, including the collision of objects and their subsequent movement.

1. Game Engine

You should implement a small game engine from scratch (based on our framework ;) that should provide the following features to the developers:

- Creating geometry and rendering the 3D scene of the game.
- Providing necessary physics simulations usable in various computer games.

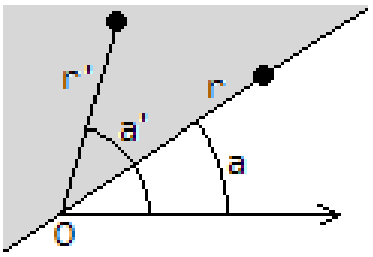
To achieve the goals, here is the specification of the minimal functionality of the engine:

- 4D vector class (optional: vectors of other dimensions)
  - Unit vector
  - Magnitude of a vector
  - Opposite vector
  - Add, subtract two vectors
  - Dot product
  - Cross product
  - (optional) Create at least one unit test for each operation.
- 4x4 matrix class (optional: matrices of other ranks)
  - Multiplication
  - Transpose
  - Multiplication by a vector.
  - (optional) Conversion of a rotation matrix to an orthonormal basis.
  - (optional) Conversion of an orthonormal basis to a rotation matrix.
  - (optional) Inverse
  - (optional) Create at least one unit test for each operation.
- AxisAngle class - represents a rotation about a given unit axis by a given angle
  - Conversion to a quaternion.
  - (optional) Create at least one unit test for each operation.
- Quaternion class
  - conjugation
  - length
  - inverse
  - dot product
  - normalize (to unit length)
  - slerp
  - Conversion to AxisAngle.

    - (optional) Conversion to rotation matrix.
    - (optional) Create at least one unit test for each operation.
- Polar coordinates (r,a), where "r" is a radius and "a" is an angle:
    - Normalization, i.e., the clipping of the "angle" coordinates to the range [0,2**π**).
    - Conversion to and from 2D Cartesian coordinate system (obtained from 3D coordinate system where the "up" axis coordinate is ignored).
    - (optional) Create at least one unit test for each operation.

----- end of Milestone (see below) -----

- Collision system:
    - In 3D/4D Cartesian system:
        - Create an algorithm computing the closest point X on a given line segment AB to a given point P.
    - In polar coordinates:
        - Create an algorithm deciding whether a point (r',a') is "on the left" (in the grey area) or "on the right"(in the white area) of the point (r,a). [In the figure, the point is "on the left"].



        - Create an algorithm computing a minimal distance (difference) between two angles in the range [0,2**π**). For example, the result for angles **π**/4 and 7**π**/4 is **π**/2.
    - (optional) Create at least one unit test for each algorithm.
- Physics:
    - Recovery from the collision - a simple bounce of a sphere from a static (kinematic) object, as shown in the slides "PA199 Collision detection in the assignment.pdf".
    - (optional) An ODE solver (see lectures); implement at least two methods (e.g. Euler and Midpoint).

Usage of 3rd party libraries: You are only allowed to use 3rd party libraries automatically installed when building the project template (available as a ZIP file in the study materials of the course). Usage of any advanced libraries such as Ogre, Havok, or OptiX is not allowed. The purpose of the assignment is to learn how to write maths, physics, and rendering on your own.

2. The Game

2.1. Motivation – Breakout game

Your goal will be to implement a "circular" variant of a classic breakout game – the goal of the game is to break all bricks in the level using a ball(s), influenced by in-game physics and paddle(s). The movement of paddles is controlled by the player.

Basic rules:

- The ball starts near the paddle; after launch, the ball moves at a constant speed.
- When the ball collides with a brick, the ball bounces, and the brick is damaged or destroyed.
- If the ball leaves the game area, the player loses one life. If any lives are left, the ball is respawned near the paddle. Otherwise, the game ends.

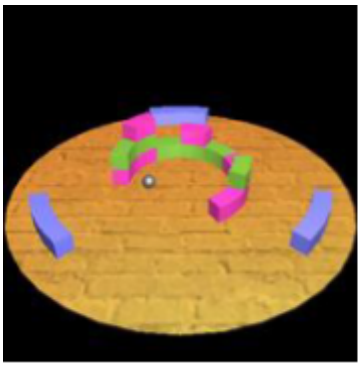
Typical extensions in real games:

- There can be multiple balls in the game at the same time. The player stays alive as long as at least one ball is inside the area.
- The paddle can change its size.
- The speed of the ball(s) is increasing over time.
- Various types of bricks exist, with various properties (durability, power-up drops, etc.)

2.2. Game specification

You should develop a "circular" breakout 3D game.

- The bricks will be arranged in a cylindrical wall in the center of the circular play area. The wall will have several rows, and each row will contain at least 12 bricks. When a brick in the bottom row is destroyed, bricks above it will fall down.
- The player will be controlling three curved paddles positioned near the outer edge of the play area.
- All paddles move simultaneously – the player can rotate them in CW and CCW directions.



2.3. Minimum functionality

All of the following have to be implemented and fully functional.

- Game mechanics
  - The player can control the paddles using the ”Left” and “Right” arrow keys and launch the ball by the spacebar.
    - Make sure the motion of paddles is smooth, i.e., they will move with an angular velocity, which must be independent of the frame rate and key repeating.
      HINT: Use the events from the keyboard (key press/repeat) to track the “pressed state” of an arrow key in a boolean variable.
    - Include the speed of the paddle in calculations of collisions with the ball. I.e., if the paddle is moving at the time of the collision, the ball will bounce in a different direction than in the case of the static paddle. The speed of the ball will not change. (see “*Collision Detection in the Assignment.ppsx*” for one possible solution)
  - “Game over”, “You win” and “Pause” messages will be shown at appropriate times:
    - When the ball leaves the play area, the “Game over” message is shown, and the game waits for the player’s input.
    - When all bricks of the wall are destroyed, the “You win” message is shown, and the game waits for the player's input.
    - During the “playtime” (the ball is moving), the player can pause the game using the “P” key. When paused, the movement of the ball and the paddles are stopped, and the “Pause” message is shown. The game can be resumed by pressing the same key again.
  - Camera switching between "perspective" and "top" views via “1” and “2” keys, respectively. The default (initial) is the "perspective" view.
- Graphics
  - Construct the geometry of the level from scratch, including:
    - ball,
    - paddles,
    - wall made of individual bricks (brick and paddle could have the same shape, just different sizes).
    - play area ground.

    All geometries must be constructed procedurally:
    - The ball and the ground must be parameterized by the radius.
    - A brick and a paddle will be parametrized by width, height, and the angle representing the length.

        - Each geometry should further be parametrized by a “level of details”. E.g., the ball could be parametrized by the angle between two vertices in horizontal and vertical directions.
        - An approach based on loading predefined 3D positions of all points from the header file is not acceptable.
    - Add colors to the ball, paddles, and bricks; (optionally, use textures)
        - Choose a fixed ambient color, fixed diffuse and specular coefficients, and the specular exponent.
    - Add texture to the ground
    - Camera in “perspective” and “top” views:
        - “Perspective”: The perspective projection must be used. Choose an appropriate angle between the up-direction of the camera and the normal of the play area - aka “pitch” rotation (hint: 45 degrees works well). The camera should not be “rolled” - i. e. the right-direction of the camera must be parallel with the play area.
        - “Top”: The orthographic projection must be used, and the camera must look at the play area from the top. The projection plane must be parallel to the play area.
        - Both views: The camera always looks at the center of the circular play area. The distance between the center of the camera and the center of the play area must be adjusted so that the rendered scene fits the window. After resizing the window, the camera should be adjusted so the whole play area still fits the window and the scene is not stretched, i.e., a sphere won't be deformed to an ellipsoid.
    - Lighting:
        - ambient, diffuse, and specular light based on the Gouraud model. (Phong is also acceptable ;)
    - Rendering the “Game over”, “You win”, and “Paused” messages: Prepare textures with the messages and render them using a quad (or a pair of triangles). Using a font-based approach is basically a no-go, as there is no library to work with fonts in the framework.
- Physics – real-time simulation of the ball movement based on collision detection and response (see Collision Detection in the Assignment.ppsx for details):
    - Ball – paddle (the curved shape of the paddle matters)
    - Ball – brick (the curved shape of the brick matters)
    - Ball – the invisible outer edge of the play area. If the last ball hits this, the “Game over” message is shown.

2.4. Above minimum functionality

Implementing each of the below-defined "extensions" will increase your grade by one level. Therefore, for an A grade, you need to implement four of the following extensions:

- **Extension 1**: life counting and scoring system:
  - gameplay-wise:
    - Counting the players' lives. The player will start with three lives. Each time the ball hits the outer edge of the play area, the player will lose one life, and the ball will be respawned to an initial position.
    - Counting score - e.g., +1 point per brick hit.
    - If the player manages to clear the whole level, the wall of bricks will be restarted; the number of remaining lives and the score will not be restarted.
  - graphics-wise:
    - Render lives at one corner of the window in the form of images (e.g., heart). For each live, one image is shown.
    - Render the score in another corner of the window. Prepare 10 textures, one for each digit 0, 1, 2, ..., 9, and render the score number as the corresponding sequence of the images. Hint: you should write a function converting an integer number (score) to individual digits.

- **Extension 2:** the durability of bricks:
  - gameplay-wise:
    - some bricks will be more durable. The ball must hit them more than once before they break up.
    - optional - the more durable the brick, the higher score the player will receive
  - graphics-wise:
    - a durable brick will change color (or texture) each time it is hit
- **Extension 3:** "power-ups" mechanics
  - gameplay-wise:
    - after breaking some bricks (could be predefined bricks or simply, e.g., each fifth brick), a power-up will appear at the position of the destroyed brick and will start moving toward the outer edge of the play-area.
    - If the player is able to "catch" the power-up by any paddle, it will be applied. Otherwise, the power-up will just vanish.
    - Possible power-ups (implement at least two types):
      - Extra life (this power-up should appear only rarely). This is dependent on "extension 1".
      - Extra score. This is also dependent on "extension 1".
      - Increase or decrease the speed of the ball.
      - Increase and decrease the size of paddles.
  - graphics-wise:
    - rendering power-ups as simple colored spheres is acceptable. Different power-ups must have different colors ;)
    - optional - use textures

        - optional - use more complicated geometry (with sphere "collider")
- **Extension 4:** "advanced" graphics (all of the following have to be implemented):
    - Rendering of bricks - when the ball is "behind" the wall, occluding bricks will be rendered as a wireframe (optional - use semi-transparency instead of wireframe) - so the ball is always visible.
    - Lighting - use the Phong model instead of Gouraud.
    - Simple cast shadow of the ball on the ground - blob shadow or other "fake" technique is also acceptable.
- **Extension 5:** "advanced" physics (all of the following have to be implemented):
    - Constantly increasing the ball's speed (e.g., speed will increase by 25% each minute).
    - "speed-up" area - if the ball reaches the centre of the game field (i.e. inside the brick wall), it will noticeably (by, e.g. 50%) increase its speed. After leaving this area, the speed will decrease to the previous speed. The changes in speed will not be instantaneous - the ball will be increasing/decreasing its speed smoothly in a reasonable timeframe, e.g. one half of a second.

3. Milestone

Create a tag named "Milestone" to the commit in your repository which will be considered as the final version for the milestone. Expected functionality:

- "Math" section
  - without "Physics"- see "line" above
- camera class
- procedural generation of geometry
  - floor, ball, paddles, bricks
- textures, lighting

Functional milestone (as declared above) will be rewarded by 1 point for the final grading.

Deliverables:

- Commit everything into GITrepository, and add the tag "Milestone" to the proper commit.
- Upload into the homework vault a compressed .zip package containing the following:
  - "**dist**" folder containing binary
  - "PA199**project**" folder containing all source codes
  - (optional) source codes of the framework, if you made any changes in the framework (outside of the project folder).

4. Final Game

The executable version of the game must fulfil these conditions:

- Be runnable on Windows OS.
- Have "reasonable" difficulty (not extra low nor high) - you should be able to play your game and win the level.
- Provide "reasonably" fast rendering. When run on "decent" HW (e.g. NVIDIA GTX 1070 or newer), the game should be rendered at least 30 FPS at FullHD resolution.
- FPS independent rendering - the speed of gameplay (speed of movement of the ball and paddles) should be independent to FPS of rendering.

The final game must have **fully** implemented mandatory requirements in sections "1. Game Engine" and "2.3 - Minimum functionality" to be accepted. It is not possible to pass the course without the final game being accepted. The accepted final game will be rewarded by 1 point for the final grading for each **fully** implemented and functional extension.

5. Repository, Devlog

Similar to other GameDev courses, mandatory parts of the project are:

- Use GIT repository we created for you at https://gitlab.fi.muni.cz/gamedev/2022/pa199
- weekly updated `devlog.md` text file:
  - short summary of work done during the week

- (optional) time spent on the project per week

6. Deadlines

- **Milestone**: **7 November 2023** (week 8).
- **Final Project**: **12 January 2024**

7. Submission

Create a folder: "PA199 Project - [your-surname-name your-first-name]" and put the following into this folder:

- Folder "**dist**", containing the runnable version of the project.
  - Make sure that both executables (the game and the tests) are present in the "dist" directory.
- Folder "**src**", containing all source codes, project file, etc. I.e., the content of the folder "`./courses/PA199/project`".
- File "readme.txt" containing a short description of the extensions you have implemented.

Pack the folder into a .zip archive and upload this archive into the homework vault in IS.