

Documentation Slide 'n' Solve

Group/Game Name: Slide 'n' Solve

Additional libraries:

- Tinyobjloader
- miniaudio

Gameplay:

Mandatory:

- 3D Geometry:
The box with front-face culling is procedurally generated at runtime and all non-trivially shaped objects (player, obstacles, goal) were modeled and textured in blender. They are .obj files and are loaded in ObjectLoader.cpp which uses the tinyobjloader library.
- Playable:
It is a strategy game.
- Min 60 FPS and Framerate Independence:
All of the movements are implemented with deltaTime and are therefore completely framerate independent. To avoid unnecessarily high framerates, the framerate is limited to 120FPS.
- Win/Lose Condition:
You have successfully solved a level when the player (yellow) hits the goal and you have failed when a movable obstacle (red) hits the goal. Once all levels (3 for each difficulty, 12 in total) are solved, the total time it took you is displayed.
- Intuitive controls:
 - Mouse: Rotate Camera
 - W: Slide forwards
 - A: Slide left
 - S: Slide backwards
 - D: Slide right
 - Shift: Slide down
 - Tab (instead of Caps Lock): Slide Up
 - E: Select "Easy" level
 - M: Select "Medium" level
 - H: Select "Hard" level
 - I: Select "Impossible" level
 - R: Restart level
 - F1: Open help page
- Intuitive Camera:
Orbit camera without strafing, Camera shakes on collision

- **Illumination model:**
All gameobjects have normals, textures and material properties. The illumination is calculated in the fragment shader using phong shading. The scene is lit by 1 directional light and 2 point lights (one inside the player, one inside the goal).
- **Textures:**
All objects have textures (loaded from a .dds file) attached to them, with mipmapping and trilinear filtering enabled. The unwrapping was done blender, the uv-coordinates are read from the OBJ-file.
- **Moving Objects:**
The player (yellow) and movable obstacles (red) can move.
- **Documentation:**
A documentation is included in the submission.
- **Adjustable Parameters:**

Optional:

- **Collision Detection (Basic Physics):**
Collision detection is an integral part of the game and is implemented in MovementManager.cpp. To ensure that the cubes stay perfectly aligned to the grid, the exact position where the cube has to end up is calculated immediately after the input (before the object even starts moving).
- **Heads-up Display:**
The current move-count is displayed as a text on the HUD and whenever something hits the goal, a feedback screen is shown ("Solved" or "Failed"). The font is loaded from a sprite atlas (the .dds file in assets/HUD) and the corresponding .fnt file. In order to display text, a quad is generated for each letter/symbol with the correct uv-coordinates, size, offset, etc. All HUDElements are rendered with the hud_pipeline and simply do not have the viewProjMatrix of the camera applied to their positions.

Effects:

Advanced Modelling:

- **GPU Particle System using Compute Shader:**
In Main.cpp a buffer with MAX_PARTICLE_COUNT dead particles is initialized (sb_particles). Whenever a movable object stops sliding (aka hits something) a new Collision is constructed and copied into the shader storage buffer sb_collisions. The compute shader particles.comp reads from that buffer and creates the particles at their correct positions with slightly randomized position, size, color and velocity. Those particles are written into the SSBO sb_particles. When alive_count hits MAX_PARTICLE_COUNT older particles are reused. The sb_particles is then passed as a vertex buffer to the vertex shader (with some of the particle-properties being ignored).

Animation:

- **Hierarchical Animation:**
The decorations on the cubes have their own meshes but they are not

GameObjects, instead they are children (see `AnimatedChild.cpp`). When a `GameObject` is rendered, all of its transformations are applied to its children. The `getLocalTransformationMatrix()` method returns different functions over time to create the rotating effect.

Shading:

- Cel Shading:

In the onscreen fragment shader there is the function `discretizeForCelShading()` which discretizes the brightness of the given color according to the given parameters. Since all objects are cube-shaped with sharp edges and there are no complex textures, this has the same effect as discretizing the output colors themselves. It gives the scene a more cartoonish look.

Post Process:

- Contours via Edge Detection:

In a new offscreen render pass before the onscreen one, the scene's normals are rendered into a designated `VkImage` (`offscreen_pass.color.image`) and the depth – in addition to being used as a depth stencil – into another one (`offscreen_pass.depth.image`). The onscreen render pass then receives these two as uniform `sampler2D`'s and in the `getEdgeFactor()` function applies a kernel to it consisting of a Sobel Filter and Gaussian Blur. One minus the maximum resulting value is then multiplied onto the shaded color, which lays the edges in black over the scene.

The depth information is linearised and then normalised, so that the point on the big surrounding cube closest to the camera gets value 0 and the one furthest away value 1.

Running edge detection on the rendered depth improves the overall result, since most faces of the meshes in the scene are axis-aligned, so in the rendered normals they fuse together with the faces of the big "scene cube", e.g. a front face of a game object has the same normal as the backface of the scene cube, even though one would expect an edge to be drawn between them. And that can be done by accounting for depth as well.

Walk-through:

Shortest solution: Left-Up-Back-Right-Down-Forward-Right-Back

How to fail: Left-Up-Back-Right-Down-Forward-Left-Back