Chroma Wars

Development Status:

Aside from the View Frustum Culling and the HUD we were able to implement every effect we wanted to have present in this game. The next steps would include the View Frustum and HUD implementation, aswell as a prettier level with better lightmaps. Also some events like the spawning of a goblin, the stealing of colour, damaging a goblin or a goblin despawning would get underlined with the help of some audio.
The objects in the game are illuminated by a directional light representing the sun. Every object has at least one texture, although most also use a second texture as a lightmap.
All the objects in the game were created using Blender.

Controls:

Key	Effect
1	Change into Attack mode
2	Change into colour return mode
W/A/S/D	Move forward/left/back/right
Move the mouse	Move the camera and change character orientation
Left mouse button	Use the action of the current mode
Esc	End the Game

Implemented Requirements:

Playable: The Game was tested on the lab computers and should be playable. The needed objects and textures are loaded, and after additional setup work the game loop starts. Every $1/60^{th}$ of a second the positions of objects and particles are updated, aswell as animations updated.
3D Geometry: The elf is our most complex 3D object, but the goblins are also not trivial. Both are loaded from a .dae file with the help of Assimp.
Win / Loose Condition: If a Goblin reduces the colour of an object to 0 the game is lost. By defeating Goblins shards can drop, collecting 3 of those will win the game.
Intuitive Controls: W A S and D Keys are usually used for in Video Games, but new players should have no problem learning the controls, since the map naturally onto the 3D Plane. Moving the Mouse rotates the camera in a sphere like fashion around a point slightly to the top right of the character, allowing easy control. Since most things on the compute can be activated using the left mouse button, activating the action with the help of it should not be hard to guess or learn. Changing action modes with the left hand is easily done because of the proximity to the W A S D keys. The Inputs are stored in an array, the entries of which encode the needed information, this array is then used every $1/60^{th}$ of a second to change variables in the game if needed.
Intuitive Camera: Like already mentioned, the camera is a spherical camera rotating around the player character, the mouse movement should translate without issue onto the sphere, allowing for smooth camera control.
Textures: Textures are usually loaded from a .dds file with the help of the INIReader from the ECG framework, in this game we use standard textures aswell as lightmaps. Also 2 textures are used to render from FBO's onto, as explained later on.
Moving Objects: Enemies, the player and particles move around the world. While particles will be explained later on the player moves with the help of the saved input, adding vectors together to then reach a final direction. The Enemies move from point to point in the game world, facing towards their new direction when the old point is reached.
Documentation: We hope this document fulfils the documentation requirements.
Adjustable Parameters: The wanted parameters can be changed by changing the contents of a .config file, which is read with the help of the INIReader from the ECG Framework.
Physics Engine: We used PhysX to detect collisions of the player or goblins, aswell as to keep them from falling through the floor, walking through walls or similar things.
Lightmaps using separate textures: For the relevant objects we created a lightmap with the help of Blender, the intensity read from those textures is then multiplied by the normal colour of the object, reducing it if it shadow falls onto it.
GPU Particle System using Compute Shader: When colour is stole or reapplied particles spawn randomly in a sphere and the fly towards their target, which is either another sphere or a point. Spawned particles are added to a vector which is used to buffer the data to a compute shader, which updates the particles position and velocities. From another buffer those updated particles are then read and used to create the visible particles before once again being updated.
GPU Vertex Skinning: The Goblins and the player are both animated. Each of them have bones which move throughput the animation, the vertices of the models include weight information, which is then used to read the needed transformation from the bones and apply the to the vertex position itself in the vertex shader.
Cel Shading: The light values of the objects are discretized in the fragment shader, resulting in a lesser smooth lightning transition.

Contours via edge detection: The objects are not directly rendered onto the screen, in a first step their colour information is rendered onto a texture with the help of a fbo, in a second step the objects normals are rendered onto another texture. In reality only a single quad is visible directly front of the screen, onto which the textures are projected. A Sobel filter is used over the R, G and B values of the normal texture, the results are combined to then read the edges from the colour image. After the 2 textures are combined accordingly the whole scene is heavily contoured. Heavy guidance was taken from https://de.mathworks.com/matlabcentral/fileexchange/28114-fast-edges-of-a-color-image-actual-color-not-converting-to-grayscale.

Used Libraries and Code:

The game uses Assimp to load the geometry data. https://github.com/assimp/assimp/releases/tag/v4.1.0
PhysX is used for collision Detection. https://github.com/NVIDIAGameWorks/PhysX/releases/tag/4.0.0
The ECG Framework library is used to load .dds Images, GLEW and GLFS are used to handle input, the window and to communicate with openGL. The Libraries were taken directly from the ECG framework available in Tuwel.
In the fragment shader we use code to transform RGB values into HSV values and back, the code is taken from the answer by “sam hocevar” on https://stackoverflow.com/questions/15095909/from-rgb-to-hsv-in-opengl-glsl