Seganku: CGUE 15S Submission 2

Johann Sebastian KIRCHNER (0926076), Nikolaus LEOPOLD (1327344)

Seganku is an animal survival hiding and foraging game inspired by 'shelter'.

The protagonist is a small skunk in need to get ready for hibernation. For that purpose they shall eat a certain amount of food to get nice and fat, and then find back to the den, before the sun sets, so they will be safe. Antagonists are eagles, from which it is advised to hide in bushes. All that running around and searching for food also uses some energy, and will require a proper ingestion of wild edibles.

The eagle circling in the sky will at times try to chase and eat the protagonist unless they manage to find cover or use their special defense technique. It's vital to keep an eye on the sky!

Controls

The player controls the skunk using 4 directional keys and mouse look (orbiting camera). For testing purposes, a free fly camera can be used alternatively.

Key	Effect	Key	Effect
W,A,S,D	Move player / free fly camera	F2	Toggle debug information
Shift	Hold down to move more quickly	F3	Toggle wireframe drawing
F	Skunk Secret Olfactory Defense	F4	Toggle texture filter mode
Mouse move	Rotate camera	F5	Toggle texture mipmap filter mode
Mouse scroll	Zoom camera (change FOV)	F6	Toggle SSAO
Tab	Toggle camera free fly mode	F7	Toggle Shadows
Q,E	Move camera up or down [free fly]	F8	Toggle View Frustum Culling
Backspace	Restart game	F9	Toggle Transparency
ESC	Quit game	F11	Display shadow map

Libraries

minimum required: C++11, GLFW 3.0, OpenGL 3.3 core, GLEW 1.10, GLM 0.9.5, Assimp 3.0, FreeImage 3.15, Bullet 2.82, FreeType 2.3.5

Note that this game is developed cross platform on Linux and Windows, the latter being the main target platform.

Implementation Details

For further documentation please refer to the generated doxygen html files. What's new since the last submission is described on the next page.

- Game Loop:

  The main game loop and the rendering pipeline structure are layed out in the main function, which calls initialization, update, draw and cleanup functions.
- Transformations:

  Transformation matrices and related interfaces are implemented in the SceneObject class. Geometry, Light and Camera are subtypes of SceneObject. Player is a subtype of Geometry.
- Model loading using Assimp:

  This is implemented in the Geometry class, which traverses the loaded assimp scene to create Geometry objects for each assimp mesh structure. A Geometry object holds a list of Surface classes, which store the actual vertex data (positions, indices, normals, uvs), communicates with the VBO and stores pointers to the used textures. Textures are referenced in the Geometry object and reused among its Surfaces.

  Note: we currently use the open standard COLLADA to store the model data. Blender is used for modelling or to prepare existing models and to export to COLLADA.
- Texture loading using FreeImage:

  The Texture class loads image files using FreeImage to generate OpenGL textures, which can then be bound to the active texture unit. Image files should have an 8 bit RGB encoding with no alpha channel. Due to the style of graphics we only load a diffuse texture for each Surface using Assimp (see above).
- Shading, Lighting and Materials:

  The Shader class implements loading, compilation and linking and usage of the source files of the GLSL shader stages. Although the architecture allows for flexible shader switching, for now we use an Ubershader approach, in that we use just one shader that provides all needed functionality for the main scene drawing, since allmost all of our objects are textured. For now a Blinn-Phong illumination model is sufficient for our needs, since it also allows for Lambertian lighting.

  The Light class is needed mostly to have dynamically changing lighting (movement and color transitions) which is needed for the day and night cycles.

  Since we only use diffuse textures mostly (specular and normal should be possible but not needed, since they don't really fit the cartoony style), basic shader uniforms and material properties are sufficient. Materials are simple GLSL structs in the shader, encapsulating the samplers, specular color and shininess exponent for the Blinn-Phong model. There is also a Light struct uniform allowing customizable ambient, diffuse and specular colors and light position.
- Low-poly graphics:

  We try to use a consistent low-poly 'cartoony' style, to avoid the uncanny valley effect. Although this is mostly just relevant for human characters, we still want to make sure to avoid any uncanny appearances. Trees, shrubs and carrots are modeled by ourselves. The skunk, eagle and cave are adapted models of a squirrel, a hawk and a cave from BlendSwap.

The following features have been added since Submission 1. Note that the gameplay details are described on the next page.

- Text Drawing:
  The FreeType library is used to generate glyph textures and the defining drawing parameters from TrueType font files which can then be used to draw text. This is implemented in the TextRenderer class.
- Physics and Collision Handling:
  Collision Handling is implemented using the Bullet Library and the Physics class. Collision Objects are the player, floor, trees and the cave walls. Bullet also takes over player movement by setting a linear velocity factor, moving the rigid body, not the model. The players location is then updated using the location of the rigid body.
  Further collision objects are carrots and bushes, they are used together with collision callbacks to detect game relevant situations (e.g. player hiding in bush, player close enough to eat carrot), they are implemented as collision-less objects though.
  All collision objects are created in the Physics class (with the exception of the players' rigid body), the floor by using the triangle mesh, carrots and bushes as spheres and trees as cylinders.
- Procedural Object Placement:
  Poisson Disk Sampling[1] is used to procedurally distribute trees, shrubs and carrots on the terrain in a manner that appears to have an aspect of natural growth but is not just pseudo-random, since the sampled positions must maintain a minimum distance to each other. The implementation is based on an article by devmag.org.za.[2]
- Shadow Mapping [Sebastian]:
  There are two versions of Shadow Mapping implemented: "normal" Shadow Mapping using an offset bias and PCF against Shadow Acne and to generate softer shadows and Variance Shadow Mapping. Both versions use different framebuffer objects and textures to ensure correct behavior. PCF and bias are implemented in the Shader and not using OpenGL functions, to enable a better level of precision.[3] [4]
  The Shadow Map is rendered using an orthographic projection – view matrix of the light position. The light position itself is changing every frame to simulate a day-cycle.
  F7 can be used to toggle between the different shadow map versions, the order being: PCF, VSM, no shadow mapping. With F11 you can toggle the rendered depth map.
- Variance Shadow Mapping [Sebastian]:
  Variance Shadow Mapping is the second implementation of shadow mapping. Whereas in normal shadow mapping only the depth value is stored in a texture, in VSM we store depth and squared depth in one texture, this allows for more effects to be applied to the texture (e.g. blurring).
  In the second stage we sample the depth texture and get the two moments we stored before and use those values and chebyshev's formula to determine the probability for a fragment to

1 http://people.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf
2 http://devmag.org.za/2009/05/03/poisson-disk-sampling/
3 http://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping
4 http://opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

be shadowed. To get even softer shadows we blur the depth texture between first and second pass.[5] [6] NOTE: At the date of submission VSM is not fully implemented yet and does not throw 100% correct shadows yet (e.g. player and carrots not shadowed, light bleeding artefacts).

- SSAO [Nikolaus]:

  A simple type of Screen Space Ambient Occlusion is implemented in the SSAOPostprocessor class, based on a tutorial by ogldev.atspace.co.uk[7]. This uses multiple render passes: First the screen colors and the closest view space positions are rendered to a Framebuffer object with texture attachments using Multiple Render Targets. The SSAO factors for each fragment are then calculated from the view space positions in the SSAO shader, by comparing the depth at randomly sampled points in a given radius to the actual depth at the sampled fragment. The more of the samples lie closer to the camera than the fragment depth, the higher the probabilty for the fragment to be occluded. To achieve nice results, the number of samples should be set to at least 64 in the init() function in main.cpp.
  The SSAO factors are multplied with the screen colors to produce the occluded colors.
  [Optionally, if SSAO blurring is enabled (F12), the SSAO factors texture is filtered using a horizontal and a vertical low pass filter. blurring currently seems to work on Linux only due to a bug, but this will be fixed]

- CPU Particles [Nikolaus]:

  The ParticleSystem class implements simple particles using glDrawArraysInstanced to simulate the special skunk olfactory defense smoke. Particle data is updated on the CPU and sent to the GPU buffers on each update call. For the particles, transparent textures are used, thus the particles have to be sorted by depth before drawing for transparency to work with the z buffer. The particles disperse and fade as their life time progresses.[8]

- View Frustum Culling:

  Bounding spheres are generated for each loaded Surface (using a simple arithmetic mean of all vertices) as center and the distance to the farthest vertex as radius. Frustum Culling is then performed in normalised device coordinates in Camera::checkSphereInFrustum.

5 http://fabiensanglard.net/shadowmappingVSM/
6 http://punkuser.net/vsm/vsm_paper.pdf
7 http://ogldev.atspace.co.uk/www/tutorial45/tutorial45.html
8 http://opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/

Gameplay Description

The player finds themselves in their den. Winter is coming, so they must eat a displayed number of carrots and then get back to the den, before the sun sets (as indicated by the day / night transition).
The world consists of a hilly terrain with procedurally placed trees, shrubs and carrots (distributed by poisson disk sampling), as well as the den (the cave).
The eagle circling in the sky will at times try to chase and eat the protagonist. Since the eagle may strike anytime, it is vital to keep a vigilant eye on the sky! To evade the attack, the player may find cover in shrubs or use their special olfactory defense technique when the eagle is close enough.
The player has the ability to run for a short duration, which will decrease the amount of collected carrots. Using the special olfactory defense technique will also reduce the amount of carrots.
If the eagle catches the player or the time runs out, the game is over. If the player manages to eat enough carrots and return to the den, they will make it through the winter.

Visible Effects

- There is one directional light source representing the sun.
- The illumination model used is simple Blinn-Phong, with customizable GLSL struct uniforms for light and material
- Text Drawing using FreeType
- Hierarchical Animation of collected carrots rotating above player
- Dynamic day and night cycle (sun movement and color transitions)
- Textured skunk, eagle, carrots, terrain, trees (bark and foliage) shrubs and cave.
- Consistent low-poly graphics to avoid 'Uncanny Valley' effect.
- Procedural placement of objects using Poisson Disk Sampling
- Two camera modes: Follow Player (default) and Free Fly. Toggled by [Tab] key. Zoom camera (change FoV) using mouse scroll.
- Collision Handling (It is not possible to pass through trees, terrain, cave. Shrubs and carrots are used as triggers on collision).
- Variance Shadow Mapping. Shadows dynamically update as the sun moves through the sky. Toggle using [F7] key.
- Screen Space Ambient Occlusion (toggle using [F6] key), if it looks bad, try to restart the game, sometimes the random sample vectors turn out unfortunate. Should look as in second screenshot (orange evening sky and cave visible)
- CPU Particles (press [F] key if you have eaten enough carrots).
- Wireframe drawing on [F3] key. Transparency on [F9] key.
- Toggle texture and mipmap filter modes using [F4] and [F5] respectively.
- Toggle View Frustum Culling on [F8] key

Refer to the next page for ingame screenshots.

Screenshots


SEGANKU
EAGLE STATE: RETREATING
PLAYER HIDDEN: 0
TIME UNTIL STARVATION: 61
FPS: 47
DELTA TIME: 21 MS
DRAWN SURFACE COUNT: 287
CARROTS: 0 / 10


SEGANKU
YOU GOT EATEN =(
CARROTS: 0 / 10