Andreas Krischke 01229362
Florian Langeder 01527111

SNOWFORM

Snowform ist ein Einzelspieler Geschicklichkeits-Spiel, in dem man einen Schneeball kontrolliert und diesen über diverse Plattformen steuert um alle Münzen einzusammeln. Enge Passagen und bewegende Plattformen erschweren den Weg und stellen Gefahren dar. Wenn der Schneeball in den Abgrund stürzt, ist das Spiel verloren.

Controls

Der Schneeball wird über die “WASD” Tasten kontrolliert. Mittels der “WASD” Tasten beginnt der Schneeball abhängig von der Kameraperspektive in die jeweilige Richtung zu rollen. Das Spiel spielt sich in einer 3rd Person Perspektive ab, bei der man die Kamera durch Mausbewegungen sphärisch um den Schneeball rotieren kann.

Tastenbelegung

Taste	**Effekt**
W, A, S, D	Schneeball kontrollieren
Maus Position	Kamera kontrollieren
Mausrad	Zoomen
Leertaste	Schneeball springen lassen
ESC	Spiel beenden

Andreas Krischke 01229362
Florian Langeder 01527111

Tastenbelegung [Debug]

Taste	Effekt
F1	Wireframe Modus togglen
F2	Face Culling togglen
F8	View Frustum Culling togglen
F10	Level abschließen [Debug]
F11	Level neu starten [Debug]
Q	Debug Container mit Anzahl gerenderter / gecullter Objekte und FPS togglen

Technical Details

Die folgenden Absätze beschreiben die Features von Snowform für den neuesten und letzten Release für die Abschlusspräsentation.

Playable

Wir haben das Programm im Vislab kompiliert und getestet. Beim Testen sind uns keine Probleme aufgefallen. Im neuesten Release gibt es drei spielbare Levels. Die FPS-Begrenzung wurde entfernt, weil sie auf manchen Rechnern nicht korrekt funktioniert hat.

3D Geometry

Wir verwenden in Snowform primär FBX Meshes, die wir mit der Importbibliothek assimp von der Festplatte laden. Neben dem Import aus einem File gibt es in unserer Engine noch weitere Möglichkeiten ein Mesh zu erstellen. In unserer Mesh-Klasse stehen folgende Methoden zur Verfügung:

- Mesh::CreateQuad
- Mesh::CreateCube
- Mesh::CreateCylinder
- Mesh::CreateSphere
- Mesh::CreateTorus
- Mesh::CreateFromFile

Andreas Krischke 01229362
Florian Langeder 01527111

Win/Lose Condition

Es gibt zwei Möglichkeiten zu verlieren:

1. Der Spieler fällt in den Abgrund.
2. Die verbleibende Zeit ist abgelaufen.

Es gibt eine Möglichkeit zu gewinnen:

1. Der Spieler sammelt alle Münzen und rollt zur Fahne.

Wenn der Spieler verliert oder gewinnt wird ein Overlay eingeblendet und ein Ergebnis-Text in der Mitte des Bildschirms angezeigt.

Intuitive Controls

Die Steuerung des Spielers erfolgt über die WASD-Tasten und die Leertaste. Das Spielerobjekt verwendet ein dynamic PhysX Rigid, auf das der Spieler Kräfte wirken lässt, um es zu bewegen. Der programmierte Controller wurde für die zweite Submission verbessert.

- Siehe ThirdPersonControllerComponent.h/.cpp

Intuitive Camera

Die Steuerung der Third-Person Kamera erfolgt derzeit direkt über Mausbewegungen. Die Kamera folgt dem Spielerobjekt automatisch und hält eine definierte Distanz ein. Die programmierte Kamera wurde für die zweite Submission verbessert. Mit einem Dreh des Mausrads lässt sich die Distanz zum Schneeball verändern.

- Siehe ThirdPersonControllerComponent.h/.cpp

Textures

Wir verwenden nur wenige Texturen, weil die meisten Modelle in der Spielwelt nur einfarbige Flächen enthalten und die Farbe in den Vertices gespeichert wird.

Folgende Objekte verwenden Texturen:

- Schneeball (Diffuse)
- Transition Overlay (UI)
- Arial Font Atlas (UI)

Für die Schneeballtexture werden Mipmaps erzeugt und trilineares Filtering aktiviert. Für Texturen, die im UI verwendet werden, werden bewusst keine Mipmaps erzeugt und trilineares Filtering deaktiviert.

Die Texture-Klasse unterstützt momentan ausschließlich das Laden von DDS-Dateien von der Festplatte:

- Siehe Texture::LoadFromFile

Moving Objects

Es befinden sich mehrere bewegende Plattformen im Spiel. Sammelbare Münzen lassen wir außerdem im Kreis rotieren.

- Siehe MovingPlatformComponent.h/.cpp und RotateComponent.h/.cpp

Adjustable Parameters

Folgende Einstellungen aus dem “settings.txt”-File werden unterstützt:

- Screen resolution (width/height): Die Auflösung vom Fenster.
- Fullscreen mode (true/false): Wenn aktiviert, dann wird das Fenster im Vollbild gestartet.
- Refresh rate: Frames pro Sekunden.
- Brightness ([0, 1]): Globale Ambient-Intensity.
- Title: Name des Fensters.
- Fov: Field of View der Kamera.
- Near: Near clipping plane der Kamera.
- Far: Far clipping plane der Kamera.

HUD

Wir haben eine eigene Userinterface-Scene, die über die Game-Scene gelegt wird.
Das Rendern von Text (Beispiel: FPS-Anzeige) und Bildern (Beispiel: Transition-Overlay) wird unterstützt.

- Siehe UserInterfaceScene.h/.cpp, TextComponent.h/.cpp und MeshRendererComponent.h/.cpp

Außerdem gibt es eine Credits-Scene, die nach dem Abschließen aller Levels geladen wird.

View-Frustum Culling

View-Frustum Culling mit Boundingboxes wird von Snowform unterstützt.

Der Algorithmus funktioniert so, dass für jedes renderbare Objekt vor dem Rendervorgang überprüft wird, ob sich seine Boundingbox innerhalb des Viewfrustums der Kamera befindet.

Mit der Taste “Q” lässt sich ein Debugcontainer im Spiel ein- oder ausblenden, in dem die Anzahl der gerenderten und gecullten Objekte angezeigt wird.

Mit der Taste “F8” lässt sich View-Frustum Culling ein- oder ausschalten.

Andreas Krischke 01229362
Florian Langeder 01527111

Da wir die Boundingbox-Berechnung für Skyboxes, Particlesystems und Texte in der 3D Welt nicht unterstützen, ist für diese Objekte eine eigene “DontCull” Option aktiviert. Das heißt, speziell diese Objekte fließen nicht in den Culling-Prozess mit ein.

Physics Engine

Wir haben PhysX für unsere Zwecke bereits vollkommen integriert.

Folgende PhysX Features verwenden wir:

- Meshshapes, Boxshapes, Sphereshapes
- Static Rigids (Für alle statischen Objekte.)
- Dynamic Rigids (Kinematische: Bewegende Plattformen; Nicht-Kinematische: Spieler)
- Collision-Detection mit Enter- und Exit-Callbacks
- Trigger-Detection mit Enter- und Exit-Callbacks
- CCD für Objekte, die sich schnell bewegen (Spieler)
- Raycasts mit Layermasks für Grounddetections (Spieler)

Die komplette Verwaltung der PhysX Szene wird von unserem PhysicsEngine-Singleton übernommen. Zum Debuggen verwenden wir den PhysX Visual Debugger. Der Code zur Anbindung wird durch Präprozessorbefehle nur im Debugmode kompiliert.

- Siehe PhysicsEngine.h/.cpp

Beleuchtung

Snowform unterstützt mehrere Directional-, Point- und Spotlights.

Wir verwenden pro Level jeweils ein direktionales Licht, um die Szene ausreichend zu beleuchten. Alle Spielobjekte bis auf Particlesystems (Schnee), Text in der 3D Welt, und Userinterface-Elemente, werden in der Szene durch ein direktionales Licht beleuchtet.

Es befinden sich keine Point- oder Spotlights in den Szenen.

- Siehe DirectionalLightComponent.h/.cpp, PointLightComponent.h/.cpp und SpotLightComponent.h/.cpp

Andreas Krischke 01229362
Florian Langeder 01527111

Directional Cascaded Shadow maps with PCF

Die Schatten im Spiel werden über direktionale cascaded Shadow maps realisiert. Dabei wird für jede der drei eingestellten Kaskaden die Szene aus Sicht der direktionalen Lichtquelle orthogonal gerendert, wobei die Kaskaden sich in ihrer Größe und Entfernung zur Kamera unterscheiden. Im Standardfragmentshader wird dann berechnet, in welche Kaskade das derzeitige Fragment fällt, und die Schatteninformation wird aus der entsprechenden Kaskadentextur ausgelesen und angewandt.

Die Kaskaden können durch den Standardfragmentshader visualisiert werden. Der Code dafür ist standardmäßig deaktiviert, kann aber in der “standard_fragment.glsl” Datei ab Zeile 272 zum Testen aktiviert werden:

Kaskade 1 (Rot)
Kaskade 2(Grün)
Kaskade 3 (Blau)

Die Kaskaden wurden so gesetzt, dass auch ohne die Aktivierung vom Visualisierungscode anhand der Schattenqualität ersichtlich ist, wo die Schnittstellen der Kaskaden 1 und 2 sind.

Eine Verbesserung der Qualität wurde durch eine hohe Texturauflösung erreicht. Ein Wegoptimieren von Peter panning durch Front-face Culling war leider nicht zu erzielen (eher im Gegenteil eine Verschlechterung), weshalb Peter panning nicht gezielt bekämpft wird, aber nur bei entfernteren Kaskaden vernachlässigbar gering auftritt.

- Siehe DirectionalLightComponent.h/.cpp

Andreas Krischke 01229362
Florian Langeder 01527111

GPU Particle System using Compute Shader

Für den fallenden Schnee in Snowform wurde ein GPU Particlesystem mit Compute Shadern realisiert.

Jedes Particlesystem besitzt einen Particlebuffer, der als SSBO (Shader Storage Buffer Object) vom Computeshader “particle_system_compute.glsl” anhand der mitgegebenen Parameter bearbeitet wird.

Die einzelnen Particle werden anschließend mit den Daten (primär Particleposition) aus dem Particlebuffer instanziert gezeichnet.

Folgende Parameter zur Aktualisierung des Particlebuffers durch den Computeshader werden unterstützt:

- Anzahl der Particle
- Minimale relative Emitposition / Maximale relative Emitposition
- Minimale Particle Velocity / Maximale Particle Velocity
- Minimale Particle Size / Maximale Particle Size
- Minimale Particle Lifetime / Maximale Particle Lifetime
- Random Seed, damit der Computeshader zufällige Bereichswerte berechnen kann

Es ist außerdem sehr einfach möglich jedes beliebige Mesh für Particles zu verwenden. Wir haben uns bei den Schneeflocken für Quads entschieden, weil es gut zu unserem Lowpoly-Stil passt:

Die “Prewarmtime” Option lässt unser Particlesystem außerdem schon vor Spielstart “warm laufen”.

- Siehe ParticleSystemComponent.h/.cpp

Andreas Krischke 01229362
Florian Langeder 01527111

Hierarchical Animation

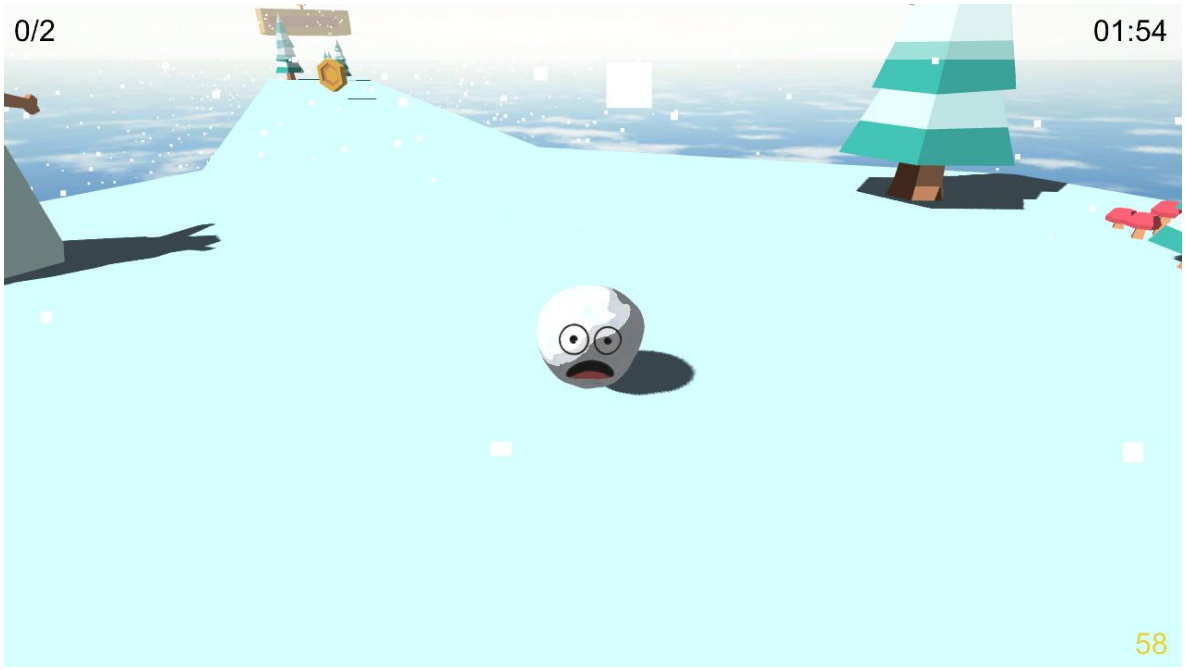
Im ersten Level befindet sich eine rotierende Münze auf einer sich bewegenden Plattform.
Durch unseren Szenengraph mit vererbten Parent-Transformationen sind solche Konstrukte relativ einfach möglich.

Specular map

Specular maps werden von unserem Standardshader unterstützt. Im neuesten Release verwendet der Schneeball nun auch seine von Paint3D exportierte Specularmap.

Cell shading

Unser Standardfragmentshader erzeugt den charakteristischen Cell shading Look in der “CalculateDirectionalLight”-Funktion ab Zeile 127. Dabei werden die Intensitäten für die Diffuse- und Specularbeleuchtung in Bereiche eingeteilt und bekommen dann den Fixwert ihres Bereichs zugewiesen:



Andreas Krischke 01229362
Florian Langeder 01527111

Weitere Features

Die folgenden Features waren keine Vorgabe, wurden aber trotzdem umgesetzt.

GameObject (Entity) <-> Componentsystem

Jedes GameObject besitzt eine Liste von Components. Wir haben die Spiellogik so gut geht in Components ausgelagert. Folgende instanzierbare Components sind momentan implementiert:

- ArcBallComponent
- DebugComponent
- BoxColliderComponent
- SphereColliderComponent
- MeshColliderComponent
- RigidDynamicComponent
- RigidStaticComponent
- CameraComponent
- MeshRendererComponent
- TextComponent
- TransformComponent
- ThirdPersonControllerComponent
- CoinComponent
- FlagComponent
- MovingPlatformComponent
- RotateComponent
- PlayerComponent
- FollowComponent
- SceneOptionComponent
- ParticleSystemComponent

Bei der Erzeugung von einem GameObject wird automatisch ein TransformComponent attached. Es existieren keine GameObjects ohne TransformComponent.

Andreas Krischke 01229362
Florian Langeder 01527111

Alle Components können folgende Events erhalten:

- **OnAwake**: Wird sofort aufgerufen, nachdem der Component instanziert wurde.
- **OnStart**: Wird beim nächsten Update-Tick aufgerufen.
- **OnAttachComponent**: Wird aufgerufen, wenn ein neuer Component auf das GameObject attached wurde. Die Referenz auf den neuen Component wird im Parameter mitgeliefert.
- **OnRemoveComponent**: Wird aufgerufen, wenn ein Component vom GameObject entfernt wurde. Die Referenz auf den entfernten Component wird im Parameter mitgeliefert.
- **OnCollisionEnter**: Wird aufgerufen, wenn ein ColliderComponent auf dem GameObject mit einem anderen ColliderComponent kollidiert ist.
- **OnCollisionExit**: Wird aufgerufen, wenn ein ColliderComponent auf dem GameObject die Kollision mit einem anderen ColliderComponent verlassen hat.
- **OnTriggerEnter**: Wird aufgerufen, wenn ein ColliderComponent auf dem GameObject in einen ColliderComponent mit Triggershape eingedrungen ist, oder umgekehrt.
- **OnTriggerExit**: Wird aufgerufen, wenn ein ColliderComponent auf dem GameObject einen anderen ColliderComponent mit Triggershape verlassen hat, oder umgekehrt.
- **Update**: Wird jeden Frame aufgerufen. Die Deltatime ist über das Time-Modul erreichbar (siehe Time.h/.cpp)
- **LateUpdate**: Wird aufgerufen, nachdem alle Update-Methoden aufgerufen wurden.
- **Render**: Wird jeden Frame nach allen LateUpdate-Methoden aufgerufen.
- **OnDestroy**: Wird aufgerufen, wenn der Component zerstört wurde.

Scenegraph

Der SceneManager-Singleton kümmert sich um die Verwaltung unserer geladenen Scenes.

Jede Scene enthält eine Liste an Root-TransformComponents, die die Basis für den Scenegraph bilden. Durch die Parent <-> Child Relationships, die durch die TransformComponents erzeugt werden, ist die Update-Reihenfolge aller GameObjects eindeutig bestimmbar.

TransformComponents mit Children sorgen außerdem für hierarchische Transformationen. Das heißt, dass automatisch Position, Rotation und Skalierung an alle TransformComponent-Children “vererbt” werden.

- Siehe TransformComponent.h/.cpp

Einen Spezialfall bilden im XML als rect=”true” markierte TransformComponents, die im XML dann eine optionale prozentuelle Angabe der “localPosition” erlauben und die Skalierung ihrer Parent-Transforms ignorieren. Diese werden ausschließlich für Userinterface-Elemente verwendet, wo eine prozentuelle Positionsangabe Sinn macht.

Laden von Szenen aus XML-Files

Alle Szenen werden bei uns komplett aus XML-Files ausgelesen und gebaut. Dieses Feature erlaubt es uns, die Level in einem anderen Programm (Unity) wesentlich einfacher und sauberer zu bauen.

- Siehe Scene::LoadFromFile

Andreas Krischke 01229362
Florian Langeder 01527111

Eigenes Textrendering

Wir verwenden keine Bibliothek zum Generieren von Text, stattdessen haben wir unseren eigenen Textbuilder programmiert, der auf Basis einer “fnt”-Datei und einer Textatlas-Textur (die beide von dem kostenlosen Programm Hiero erstellt werden) ein Textmesh erzeugt, das dann entweder im 3D- oder 2D-Raum über den MeshRendererComponent gerendert werden kann.

Unser TextComponent verfügt über folgende Optionen:

- **Enabled**: Aktiviert/Deaktiviert das Rendern des Texts.
- **Material**: Das Material (und Shader), das zum Rendern verwendet werden soll.
- **Font**: Schriftart, die zum Erstellen verwendet werden soll.
- **Font Size**: Größe, in der das TextMesh gerendert werden soll.
- **Text Alignment (TopLeft, CenterCenter, ...)**: An welcher Position der Text gerendert werden.
- **Text Overflowmode (Overflow, OverflowH, OverflowV, Ellipsis, Truncate)**: Wie der Text behandelt werden soll, wenn er über MaxWidth/MaxHeight hinaus geht.
- **Wordwrapping (true/false)**: Wenn aktiviert, werden Wörter nicht mehr abgeschnitten und stattdessen in die nächste Zeile verschoben.
- **Max Width**: Maximale Breite des Textes
- **Max Height**: Maximale Höhe des Textes

AssetManager und Loader

Alle Assets (bis auf die PhysX Objekte, die von PhysicsEngine verwaltet werden) werden vom AssetManager-Singleton verwaltet. Es ist damit möglich, zu jeder Zeit von überall im Programm Assets zu laden, zu finden oder wieder zu entfernen. Jedes Asset bekommt einen Namen, der eine eindeutige Identifikation in innerhalb der Asset-Gruppe ermöglicht.

Folgende Assets werden im AssetManager verwaltet:

- Shaders
- ShaderPrograms
- Texture2Ds
- Meshes
- Materials
- Fonts
- Cubemaps

- Siehe AssetManager.h/.cpp

Die Methoden zur Erzeugung der obigen Assets werden in AssetLoader-Objekten aufgerufen.
Ein Assetloader ist eine Klasse, die eine Load() und Unload() Funktion implementiert und dort alle für die Szene relevanten Assets ladet oder entladet.

Wir unterstützen es zwar, aber momentan verwenden wir keine eigenen Assetloader pro Szene, sondern für alle Szenen einen einzigen globalen.

- Siehe GlobalAssetLoader.h/.cpp

Andreas Krischke 01229362
Florian Langeder 01527111

Skybox

Jeder Level hat eine Skybox, die sich in jedem Level individuell einstellen lässt.

Für Submission 2 verwenden wir die von Einführung in die Computergraphik zur Verfügung gestellten Skybox-Texturen.

- Siehe Cubemap.h/.cpp und SkyboxMaterial.h/.cpp

Eingebundene Bibliotheken

- Assimp: Zum Laden von Meshinformationen.
  https://github.com/assimp/assimp
- TinyXML2: Zum Laden von den XML-Dateien für die Szenen.
  https://github.com/leethomason/tinyxml2
- PhysX: Zur Physics-Simulation, Kollisiondetection, etc.
  https://github.com/NVIDIAGameWorks/PhysX

Verwendete Tools

- Paint3D: Für die Erstellung des Schneeballmodells und dessen Textur
- Unity: Für die Erstellung der Levelszenen, die dann durch einen von uns geschriebenen Szenenexporter in ein XML-File geschrieben werden.
- DDS Converter 1.4: Zum Konvertieren unserer Texturen/Bilder in das dds-Format.

Assets (Modelle)

- **https://opengameart.org/users/kenney**
- **https://opengameart.org/content/platformer-kit**