Jonas Dietrich

Matrikel-Nr.: 407414

Game Engineering Lab

Fakultät: Informatik - Game Engineering

"Adventure Island"

Winterinsel

Grundkonzept

Meine zentrale Arbeit in der Wintergruppe des Spiels „Adventure Island“ bestand darin die gesamte Inselszenerie zu modellieren, sowie das das „Labyrinthrätsel“ zu programmieren. Die winterliche Insel beinhaltet ein Haupträtsel, welches daraus besteht, fünf Äste zu sammeln um daraus einen Eisschlüssel zu erschaffen, mit welchem man die Insel verlassen kann und das Spiel abschließt. Jeder dieser fünf Stöcke erhält man durch das Lösen eines untergeordneten Rätsels, zu denen auch das Labyrinth zählt.

Bei der gemeinsamen Ideenfindung für mögliche Rätsel bedienten wir uns an unseren eigenen Assoziationen mit der Jahreszeit Winter, anstatt Gedanken aus anderen Spielen zu übernehmen oder gar zu kopieren. Diese Gedanken hielten wir visuell fest.

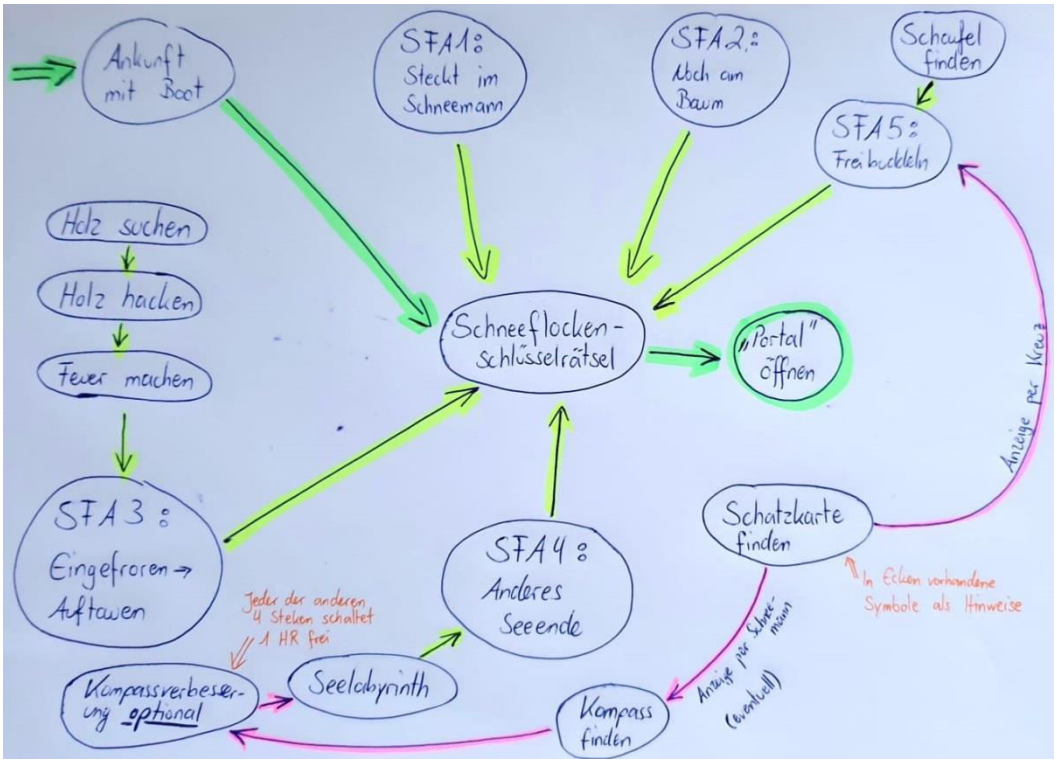


Abb. 1 – Prototypische Mindmap

Des Weiteren beschlossen wir, dass die Insel sowohl ein eher natürliches Erscheinungsbild, als auch eine Brise Magie mit sich bringen soll. Diese Informationen berücksichtigte ich und verwendete sie als Grundlage für meine Arbeit, welche ich konkret im „Making-Of“ –Segment ausführe. Für die Modellierung rundlicher Objekte einigten wir uns auf die einheitliche Verwendung von Fünf- und Siebenecken. Wir empfanden die weit verbreiteten gleichmäßigen Sechs- und Achtecke als zu generisch und geometrisch konstruiert, weshalb wir jenes alternative Kriterium festlegten.

Making-Of

1. Ursprüngliches Design und Modellierung

Zur Modellierung und Texturierung sämtlicher 3D-Objekte im Verlauf meiner Arbeit nutzte ich die Software Blender.

1.1 Winterliche Insel

Design

Als Grundlage für die Modellierung der Insel entwarf ich eine Skizze, die Grenzen und Gebiete deutlich erkennbar macht. Die obere und linke Hälfte der Insel ist ein lichtdurchfluteter, weitflächiger Bereich. Dem Spieler, welcher auf dem Steg startet, wird die Mögkichkeit geboten sich frei zu bewegen und die Umgebung zu erkunden. Durch den Verlauf der Landschaft kann er die Hütte nicht übersehen und wird bereits auf erste Objekte aufmerksam, mit denen er interagieren kann. In diesem offenen Areal kann der Spieler drei der fünf möglichen Äste ergattern, wobei er sich wie in einer Open-World frei fühlen soll und nur Klippen oder Felsen für ihn ein Hindernis darstellen.

Der rechte, untere Teil der Insel hebt sich konstratreich zum anderen Spielbereich hervor. Hier muss der Spieler durch eine dunkle, lineare Schlucht navigieren, die ihn zu einem Gebirgsee führt. Das Rätsel welches er am See vorfindet wird umso leichter, je mehr Äste er bereits davor auffinden konnte. Er soll dieses also möglichst spät entdecken, weshalb, dieses Areal durch sein Design stark vom Rest der Karte abgegrenzt ist. Darüber hinaus sorgt dieser Aufbau für optische Frische und Abwechslung.

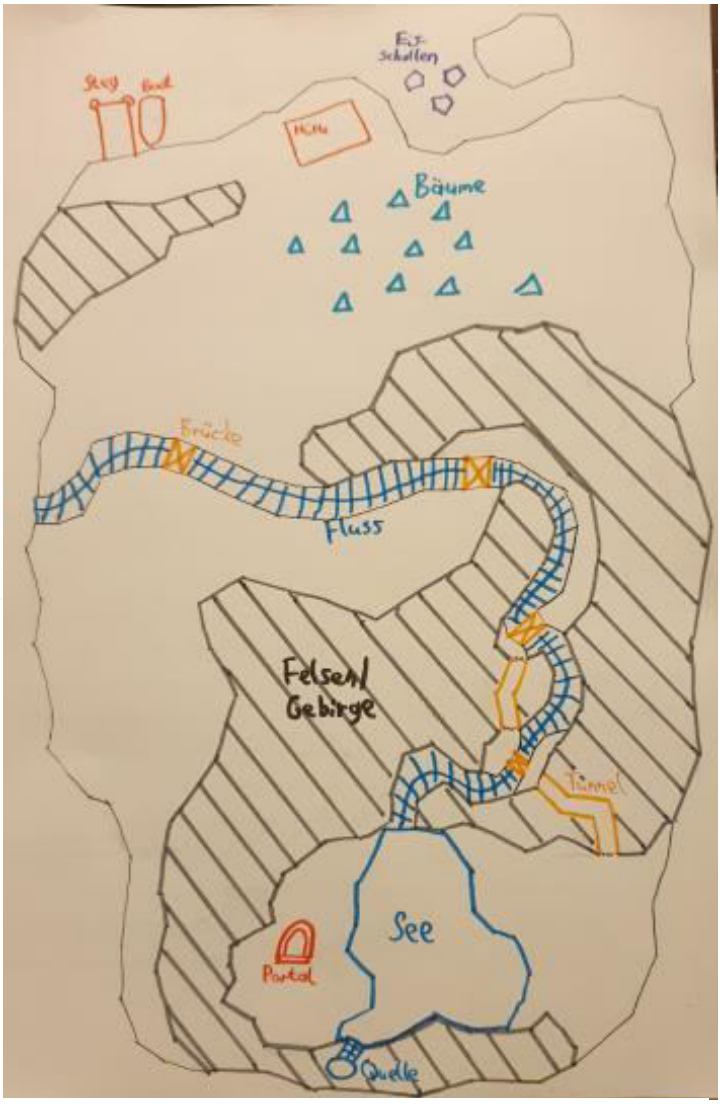


Abb. 2 – Prototypische Inselskizze aus der Vogelperspektive

Nach Rücksprache mit dem Rest der Gruppe wurde dieses Design unverändert für gelungen befunden und ich begann mit der Modellierung der Insel.

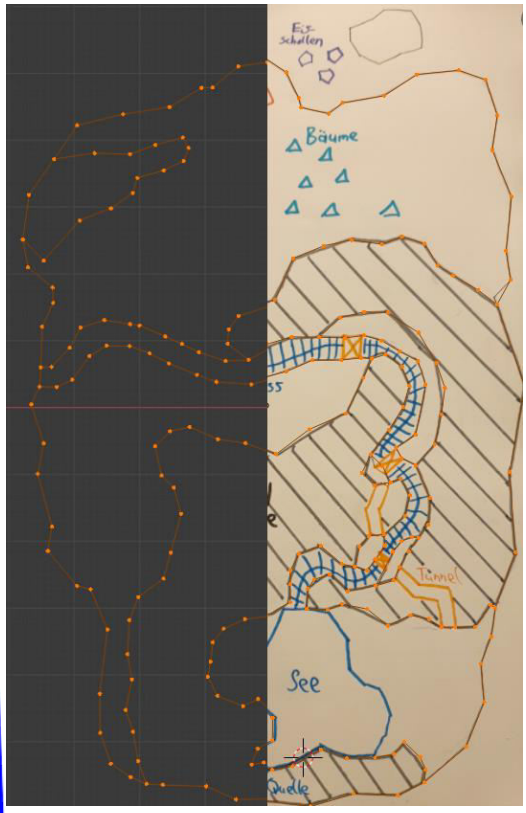
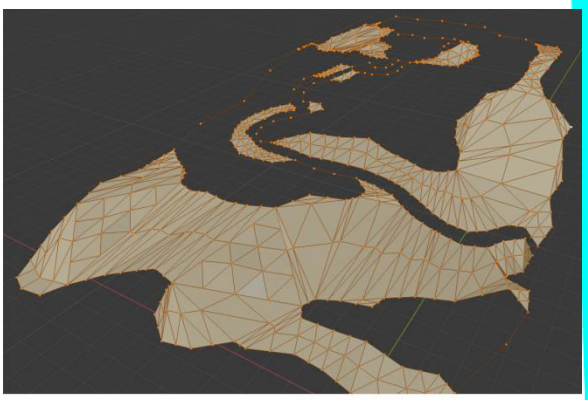


Abb. 3 – Kantennetz der Insel (Blender)

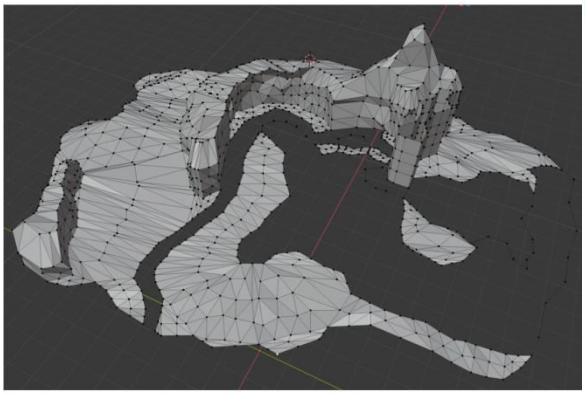
Modellierung

Im ersten Schritt extrudierte ich einen Vertex wiederholte Male entlang der Außenkanten der Insel. Im Anschluss daran extrudierte ich entlang sämtlicher formdefinierender Kanten (Gebirge, Felsen und Fluss), sodass als Ergebnis ein planares Kantennetz des Inselumrisses vorlag. Dabei achtete ich darauf, dass zwar alle notwendigen Kanten sichtbar waren, die Anzahl an Vertices dennoch so gering wie möglich gehalten wurde, um den Charme des „Low Poly"-Stils beizubehalten. Mit Hilfe der orthografischen Ansicht konnte ich exakt das gewünschte Ergebnis ohne perspektivische Verzerrung erlangen. (Abb. 3)

1



2



3

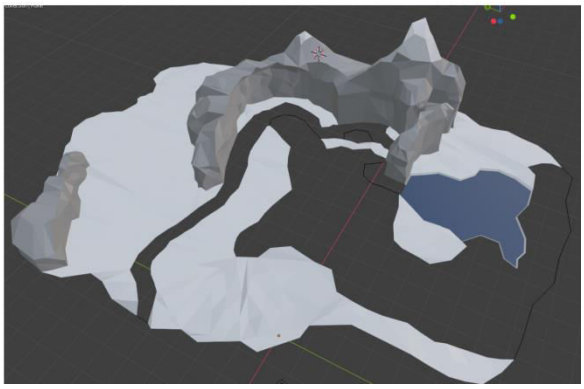


Abb. 4 – Modellierung des Terrains

Durch die gruppenweise Verschiebung der erzeugten Vertices erschuf ich schrittweise ein angemessenes Höhenprofil des begehbaren Terrains. Auch hierbei nützte mir die orthografische Ansicht für die notwendige Präzision. Dem folgend füllte ich die geschlossenen Kanten mit einem nicht planaren Face und triangulierte dieses unmittelbar, sodass sich eine fehlerfreie Struktur des Meshes ergab. (Abb. 4, 1)

Im nächsten Schritt modellierte ich die Felsen. Hierzu extrudierte ich entlang des Felsenumriss nach oben und unterteilte diese Faces mit einem Loop Cut horizontal in mehrere Segmente, welche ich gruppenweise skalierte. Hierbei wurden weiter oben liegende Faces kleiner skaliert als die darunter liegenden, sodass der Felsen nach oben hin schmal zuläuft. (Abb. 4, 2)

Ich erstellte Texturen in Form von einfachen, einfarbigen Blender-Materialien, sodass man diese direkt in Unity importieren konnte. (Abb. 4, 3)

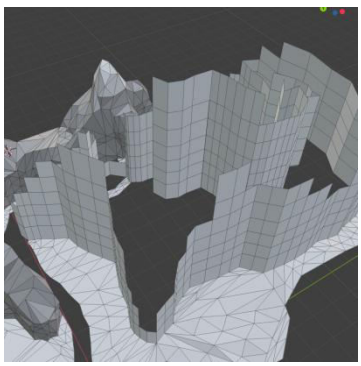
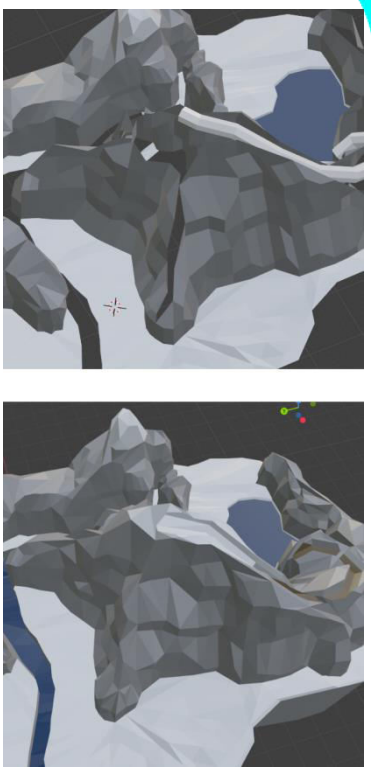


Abb. 5 – Grundgerüst des Bergmodells

Bei der Erstellung des großen Berges im Zentrum der Insel ging ich ähnlich vor wie im obigen Schritt. Nach dem Loop Cut der Faces habe ich einige Faces gelöscht, sodass ein starkes Gefälle vorherrscht, aus welchem später der Gebirgkamm entsteht. (Abb. 5)



Das Skalieren einzelner Gruppen an Faces erwies sich hier oftmals als unpraktisch, da der Verlauf des Berges infolge dessen unnatürlich ausah, weshalb ich viele Vertices manuell transliert habe.

Ursprünglich nicht im Konzept eingeplant, entschied ich mich in Rücksprache mit meinem Team dazu, den Berg begehbar zu machen und einen der Äste als in das Gipfelkreuz zu integrieren. Auf diese Weise wird nicht nur unzugängliches Terrain in zusätzliche Spielfläche verwandelt, sondern der Spieler wird für seine Bemühungen mit einen Ausblick über die gesamte Insel belohnt. Den Pfad zum Erklimmen

Um den Bergpfad zu modellieren habe ich ein planeres Vieleck entlang einer Bézierkurve extrudiert, welches ich im Anschluss in ein festes Mesh umwandelte. Diese habe ich auf dem Berg platziert und mittels dem Boolean-Modifier „difference“ die Einkerbung erzeugt, welche den Weg darstellt. (Abb. 6)

Abb. 6 – Prozess der Bergmodellierung

Abb. 7 – Insel mit dekorativen Elementen

Die erste Version der Winterinsel war nun vollendet, sodass ich sämtliche andere Modelle, sowohl von mir, als auch von meinen Teammitgliedern, in die Inselszene importierte, skalierte und diese an ihre vorgesehene Position platzierte. Unter Zuhilfenahme eines Partikelsytems wurde in Blender ein Wald erstellt, welcher jedoch durch in Unity durch ein eigenes solches System erzeugt werden soll. (Abb. 7)

Die Insel wurde zu einer .fbx-Datei konvertiert und in eine Unity-Testszene eingefügt. Hier wurde ausgiebig die Optik, die Begehbarkeit und die Kollisionen gestestet. Aufgrund der ermittelten optischen Defizite (zu dunkle Texturen, unpassend wirkende Faces, nicht sichtare Teile von Meshes) und Kollisionsproblemen (Softlocks, nicht überwindbare Passagen, zu kleine Skalierung) widmete ich mich der Fehlerbehebung. Hierzu wurden iterativ Fehler in Blender ausgebessert, die neue Datei in Unity geladen, auf Fehler geprüft und erneut in Blender ausgebessert bis das Endresultat fehlerfrei.

Hierzu gehörten: (Syntax: Fehlerbehebung -> erzielte Wirkung)

- Anpassen von Blender-Materialien -> Optische Aufwertung und Stimmigkeit der Szene
- „Beveln“ (abrunden) rechtwinkelig aufeinander zulaufender Faces -> Optische Aufwertung und weicherer Schattenübergang
- Ändern des Einstrahlwinkels von Parallellichtqullen -> Bessere globale Beleuchtung sowie leichterte Orientierung

- Finden und „mergen“ (zusammenfügen) doppelter Vertices und Faces -> Vermeidung von Renderfehlern + Softlocks
- Manuelles Verschieben einzelner Faces und Vertices -> Vermeidung von Softlocks
- Reskalieren einzelner Objekte -> Begehbarkeit gewährleisten
- Ändern der Dreickeckstruktur begehbarer Faces in eher gleichwinkiger Dreiecke -> Stimmigere Optik (Dreiecke mit zwei langen und einer sehr kurzen Kante wirkten künstlich und zu markant) (Abb. 8)
- Invertieren und neu kalkulieren von Normalen -> Korrektes Rendering von Objekten in Unity (Abb. 9)

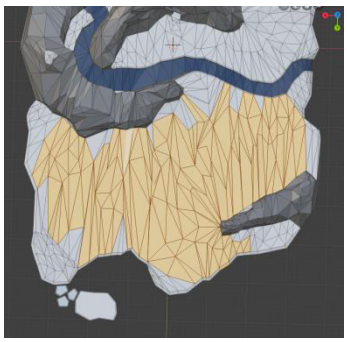


Abb. 8 – Angepasste Dreieckstruktur

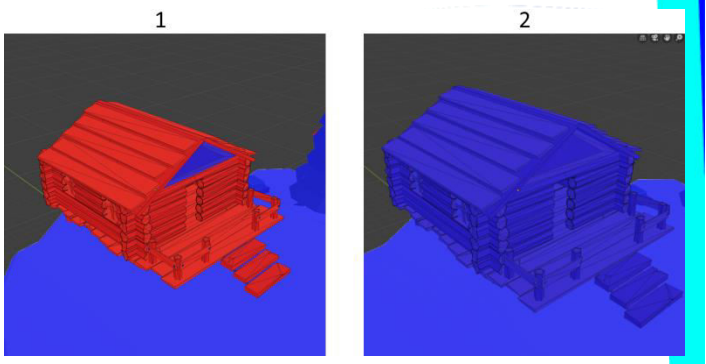


Abb. 9 – Bearbeiten der Normalen

1.2 Weitere Modelle

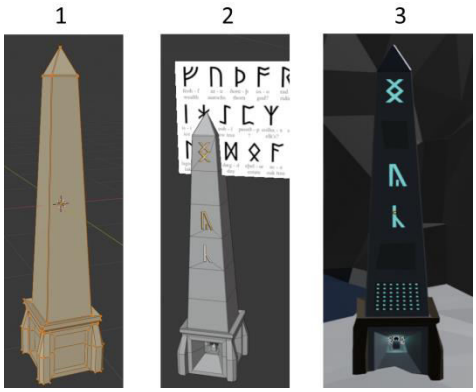


Abb. 10 - Obelisk

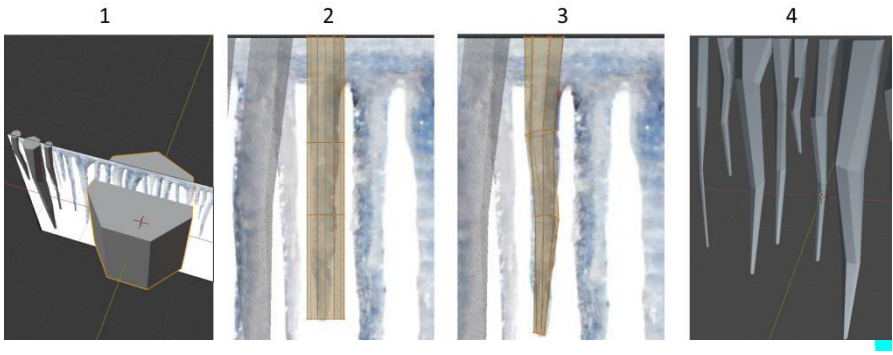


Abb. 11 – Prozess der Eiszapfenmodellierung

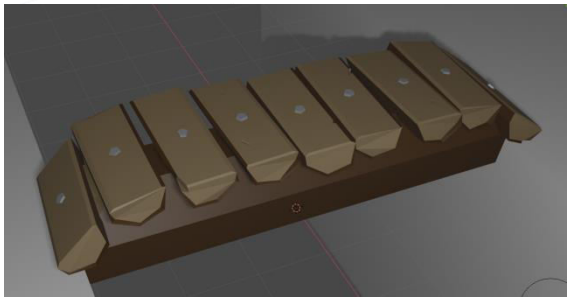


Abb. 12 – Holzbrücke

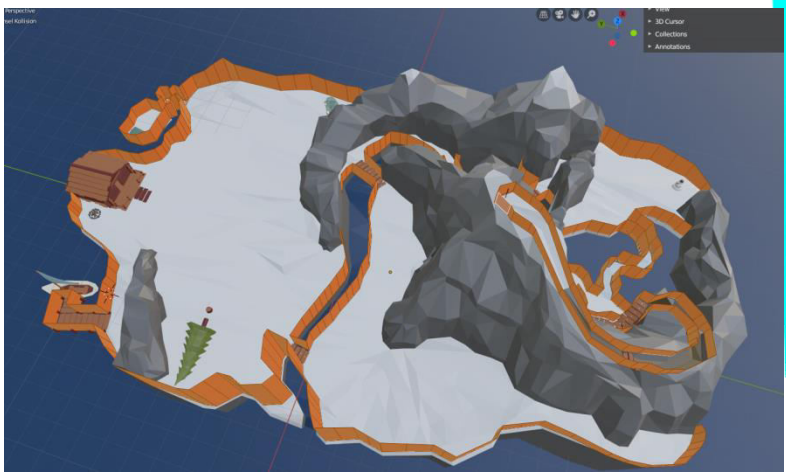


Abb. 13 – Kollisionsbarriere für den Spieler (unsichtbar in Unity)

2. Programmierung

Logik des Labyrinthrätsels: Der Spieler kann, ausgehend von den blau markierten Feldern (clicked = true), sich schrittweise durch das Labyrinth vorarbeiten, indem er auf zu andere Felder klickt. Anklickbare (zu blauen benachbarte) Felder werden grün, die restlichen Felder rot hervorgehoben wenn der Spieler die Maus darauf richtet. Klickt der Spieler auf ein richtiges Feld (fragile = false), färbt sich dieses blau (clicked = false wird zu true), klickt er auf ein falsches (fragile = true) werden alle bereits angeklickten Felder in ihren Ursprungszustand zurückgesetzt. Erreicht der Spieler den linken Rand des Labyrinths, ist das Rätsel gelöst und es wird beendet. Durch klicken auf die Zurück-Taste wird das Rätsel im derzeitigen Zustand verlassen. Hat Spieler den Kompass gefunden und bereits mindestens einen Ast gesammelt, werden ihm durch das Aufleuchten des Kompass in orange benachbarte Felder angezeigt, auf die nicht geklickt werden sollte.

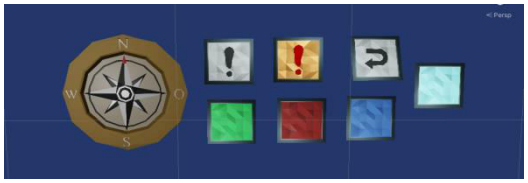


Abb. 14 – Verschiedene selbsterstellte Icons des Rätsels

Bei der Klasse „Grid Manager“ handelt es sich um einen Empty und die Parent-Klasse der einzelnen 2D-Sprites, der das gitterförmige Labyrinth in der Methode „Start()“ initialisiert. Dabei wird jedes 2D-Sprite mithilfe einer doppelten for-Schleife an seine richtige x- und z-Koordinate transliert, sowie in einem zweidimensionalen Array abgespeichert, sodass spätere Zugriffe in Abhängigkeit von relativen Positionen leicht erfolgen können. Darüber hinaus werden die Felder einheitlich benannt und über „SetFragileValue()“ für jedes Feld der entsprechende Wert für „bool clicked“ und „bool fragile“ festgelegt. Von dieser Methode wird erneut in „Update()“ zum Zurücksetzen des Rätsels. (Abb. 15)

```
Unity-Skript | 2 Verweise
public class GridManager : MonoBehaviour
{
    public int totalColumns = 13; /*Größe in X-Richtung*/ public int totalRows = 15; //Größe in Z-Richtung
    public float tileSize = (float)1.1;
    public GameObject[,] tileArray;
    public GameObject backButton;
    public Transform gridTransformer;
    Unity-Nachricht | 0 Verweise
    void Start()
    {
        Debug.Log("Start Grid Manager");
        GenerateGrid();
    }
    Unity-Nachricht | 0 Verweise
    private void Update()
    {
        if (!tileArray[12, 14].GetComponent<IceFieldBehaviour>().clicked) //Werte nur setzen, wenn nich nicht geschehen
        {
            SetFragileValues();
            Debug.Log("Fragile Werte gesetzt");
        }
    }
    1 Verweis
    private void GenerateGrid()
    {
        tileArray = new GameObject[totalColumns, totalRows]; //2D-Array zum Speichern gleichartiger 2D-Sprite-Objekte
        for (int row = 0; row < totalRows; row++)
        {
            for (int col = 0; col < totalColumns; col++)
            {
                tileArray[col, row] = (GameObject)Instantiate(Resources.Load("IceField"), gridTransformer); //"IceField" ist ein 2D-Sprite auf dem das "IceFieldBehaviour"-Skript liegt
                tileArray[col, row].GetComponent<IceFieldBehaviour>().SetPosXZ(col, row);
                float posX = col * tileSize;
                float posZ = row * tileSize;
                tileArray[col, row].transform.position = new Vector3(posX, -30, posZ); //Positioneriung in Abhängikeit der tileSize und der Position im Array
                tileArray[col, row].name = "tile_x" + col.ToString() + "z" + row.ToString(); //"IceField(Clone)" wird umbenannt in z.B."tile_x2z3", wenn gilt col = 2 und row = 3
                Debug.Log("Instantiated " + tileArray[col, row].name);
            }
        }
        float gridWidth = totalRows * tileSize;
        float gridHeight = totalColumns * tileSize;
        gridTransformer.transform.position = new Vector3(-(gridWidth + tileSize) / 2, -30, -(gridHeight + tileSize) / 2); //Zentrieren des Gitters
        Debug.Log("Aligned Ice Field Array in Center Position");
        backButton = (GameObject)Instantiate(Resources.Load("BlackArrow"));
        backButton.name = "tile_back_button";
        backButton.transform.position = tileArray[12, 5].transform.position + new Vector3(3.3f, 0, 0);
    }
```

Abb. 15 – Code-Ausschnitt Klasse „Grid Manager“

Die Klasse „MazeCameraRaycast“ prüft mittels eines Strahls von der aktiven Kamera über dem Rätsel zur derzeitigen Mausposition, ob der Strahl auf ein 2D-Sprite trifft. Ist dies der Fall wird die Methode „MouseInteraction(bool b)“ aufgerufen, welche die Spiellogik hauptsächlich steuert. Hierbei wird unterschieden zwischen Mausklick (b = true) und Mauszeigen (b = false). (Abb. 16)

```
Unity-Skript | 0 Verweise
public class MazeCameraRaycast : MonoBehaviour
{
    public float rayLength = 100;
    //public LayerMask layermask;

    Unity-Nachricht | 0 Verweise
    private void Update()
    {
        RaycastHit mouseScreenPosHit;
        Ray rayToMouseScreenPos = Camera.main.ScreenPointToRay(Input.mousePosition);
        if (Physics.Raycast(rayToMouseScreenPos, out mouseScreenPosHit, rayLength))
        {
            Debug.Log("Selected: " + mouseScreenPosHit.collider.gameObject.name);
            if (mouseScreenPosHit.collider.gameObject.name == "tile_back_button")
            {
                if (Input.GetMouseButtonDown(0))
                {
                    mouseScreenPosHit.collider.GetComponent<BackButton>().MouseInteraction();
                }
            }
            else
            {
                if (Input.GetMouseButtonDown(0))
                {
                    mouseScreenPosHit.collider.GetComponent<IceFieldBehaviour>().MouseInteraction(true);
                }
                else
                {
                    mouseScreenPosHit.collider.GetComponent<IceFieldBehaviour>().MouseInteraction(false);
                }
            }
        }
    }
}
```

Abb. 16 – Klasse „MazeCameraRaycast“

Auf jedem klickbaren 2D-Sprite liegt das „IceFieldBehaviour“-Skript, welches die Spiellogik verwaltet. In der "Start()“-Methode werden einige andere Klassen eingebunden, mit denen später interagiert wird. Zudem werden Getter- und Setter-Methoden deklariert. (Abb. 17)

```
Unity-Skript | 99+ Verweise
public class IceFieldBehaviour : MonoBehaviour
{
    public bool fragile;
    public bool clicked;
    public int positionX;
    public int positionZ;
    private GridManager gridManager;
    private GameObject fieldStatusDisplay;
    private GameObject compassDisplay;
    private GameObject hilfeVars;
    private GameObject firstPersonPlayer;
    private GameObject lakeMaze;
    private GameObject obeliskInteraction;

    Unity-Nachricht | 0 Verweise
    void Start()
    {
        fragile = true;
        clicked = false;
        gridManager = GameObject.Find("GridHolder").GetComponent<GridManager>()
        hilfeVars = GameObject.Find("ScriptVariablen");
        firstPersonPlayer = GameObject.Find("First Person Player");
        lakeMaze = GameObject.Find("LakeMaze");
        obeliskInteraction = GameObject.Find("ObeliskEmpty");
    }
    Unity-Nachricht | 0 Verweise
    void Update()
    {

    }
    0 Verweise
    public int GetPosX()
    {
        return positionX;
    }
    0 Verweise
    public int GetPosZ()
    {
        return positionZ;
    }
    1 Verweis
    public void SetPosXZ(int x, int z)
    {
        positionX = x;
        positionZ = z;
    }
```

Abb. 17 – Klasse „IceFieldBehaviour“ Teil 1

Die Methode „MouseInteraction(bool b)“ wird stets auf dem Feld aufgerufen, auf das die Maus zeigt bzw. auf das die Maus geklickt hat. Falls auf das Feld noch nicht geklickt wurde, prüft das Skript bei allen vier benachbarten Feldern (darunter, darüber, links und rechts davon), ob auf diese schon geklickt wurde. (Abb. 18, Z. 64 – 83). Sollte dies der Fall sein, wird „bool clickedAdjacent“ auf „true“ gesetzt und mithilfe „FieldStatusDisplay(2)“ wird ein grünes Feld erzeugt, das dem Spieler symbolisiert, er kann auf dieses Feld klicken. (Abb. 18, Z. 70, 74, 78, 82, 84-86) Um diese Abfragen durchzuführen, werden Felder im Array in Abhängigkeit des ausgewählten Feldes angesprochen. Zur Gewährleistung, dass kein Feld angesprochen wird, dass gar nicht in dem Array existiert, werden if-Abfragen als Vorbedingung eingesetzt. (Abb. 18, Z. 68, 72, 76, 80)

Für den Fall, dass das Feld anklickbar ist, darauf geklickt wurde und es fragil ist, wird die Methode zum Zurücksetzen des Labyrniths aktiviert. (Abb. 18, Z. 87-91) Darüber hinaus werden etwaige noch existierende „fieldStatusDisplay“-Objekte (die temporären Sprites, die dem Spieler den aktuellen

Zustand des Feldes signalisieren) zerstört, indem einmal mit zwei for-Schleifen durch das ganze Array gelaufen wird. (Abb. 18, Z. 92-101).

Wurde auf das Feld geklickt und und es handelt sich um ein Feld, welches nicht fragil ist, so wird der Wert „bool clicked“ auf „true“ gesetzt. (Abb. 18, Z. 104-106) Sollte das angeklickte Feld die Koordinaten x=0 und z=4 besitzen (das Endfeld des Labyrinths), so wird das Rätsel beendet. (Abb. 18, Z. 107-110). Hierbei werden Objekte im „firstPersonPlayer“-Empty (der steuerbare Spielcharakter) wieder aktviert, Objekte im „lakeMaze“-Empty deaktiviert (das gesamte Rätsel). Darüber hinaus wird der Mauscursor ausgschalten und der Spieler wird an eine bestimmte Stelle positioniert. Im Folgenden wird eine Animation gestartet, welche den zugefrorenen Eissee in der Spielwelt verschwinden lässt und diesen begehbar macht. (Abb. 19, Z. 110-123)

```
64  public void MouseInteraction(bool mouseClicked)
65  { if (clicked == false) //Prüfen benachbarter Felder nur, wenn auf das ausgewählte Feld noch nicht geklickt wurde
66      {
67          bool adjacentClickable = false;
68          if (positionX > 0) //Ausnahme linker Rand (X=0)
69          {
70              adjacentClickable = gridManager.tileArray[positionX - 1, positionZ].GetComponent<IceFieldBehaviour>().GetClicked(); //Feld links geklickt?
71          }
72          if (positionX < 12 && !adjacentClickable) //Ausnahme rechter Rand (X=12)
73          {
74              adjacentClickable = gridManager.tileArray[positionX + 1, positionZ].GetComponent<IceFieldBehaviour>().GetClicked(); //Feld rechts geklickt?
75          }
76          if (positionZ > 0 && !adjacentClickable) //Ausnahme unterer Rand (Z=0)
77          {
78              adjacentClickable = gridManager.tileArray[positionX, positionZ - 1].GetComponent<IceFieldBehaviour>().GetClicked(); //Feld unten geklickt?
79          }
80          if (positionZ < 14 && !adjacentClickable) //Ausnahme oberer Rannd (Z=14)
81          {
82              adjacentClickable = gridManager.tileArray[positionX, positionZ + 1].GetComponent<IceFieldBehaviour>().GetClicked(); //Feld oben geklickt?
83          }
84          if (adjacentClickable == true) //Auf Feld kann geklickt werden, da ein Nachbarfeld bereits angeklickt wurde
85          {
86              FieldStatusDisplayDraw(2); //Grünes Feld erzeugen
87              if (mouseClicked)
88              {
89                  if (fragile)
90                  {
91                      gridManager.SetFragileValues(); //Zurücksetzen in Anfangszustand nach falschem Klick (clicked + fragile)
92                      for (int col = 0; col <= 12; col++)
93                      {
94                          for (int row = 0; row <=14; row++) //Zurücksetzen der FieldStatusDisplay Overlay Bilder
95                          {
96                              if(gridManager.tileArray[col, row].GetComponent<IceFieldBehaviour>().fieldStatusDisplay)
97                              {
98                                  Destroy(gridManager.tileArray[col, row].GetComponent<IceFieldBehaviour>().fieldStatusDisplay);
99                              }
100                             gridManager.tileArray[col, row].GetComponent<IceFieldBehaviour>().FieldStatusDisplayDraw(0);
101                         }
102                     }
103                 }
104                 else //Status ändern bei erfolgreichem Klick
105                 {
106                     clicked = true;
107                     if (positionX == 0 && positionZ == 4) //Beenden des Rätsels, wenn auf das Endfeld geklickt wurde
108                     {
109                         Destroy(obeliskInteraction.GetComponent<SphereCollider>());
110                         foreach (Transform child in firstPersonPlayer.transform)
```

Abb. 18 – Klasse „IceFieldBehaviour“ Teil 2

Wurde auf ein Feld bereits geklickt wird dies durch ein blaues 2D-Sprite angezeigt. (Abb. 19, Z. 134-139) Daraufhin wird geprüft wie viele Äste der Spieler gesammelt hat und gegebenenfalls der Hilfskompass farbig hervorgehoben. (Abb. 19, Z. 140+) Auch hierbei wird der Status benachbarter Felder analog wie im obigen Code abgefragt. Die „compassDisplayDraw“-Methode hebt dabei den Kompass farbig hervor.

```
110                         foreach (Transform child in firstPersonPlayer.transform)
111                         {
112                             child.gameObject.SetActive(true);
113                         }
114                         foreach (Transform child in lakeMaze.transform)
115                         {
116                             child.gameObject.SetActive(false);
117                         }
118                         Cursor.lockState = CursorLockMode.Locked; //Maus wird wieder deaktiviert
119                         Cursor.visible = false;
120                         firstPersonPlayer.transform.position = new Vector3(-45.8f, 27.5f, 93.2f);
121                         Quaternion playerRotation = Quaternion.Euler(0, 78.6f, 0);
122                         firstPersonPlayer.transform.rotation = playerRotation; //Spieler so platzieren, dass er den See sieht
123                         hilfeVars.GetComponent<HilfeVars>().LakeLerp = true; //Lerpen der zugefrorenen See starten
124                         Debug.Log("Lake Lerp: " + hilfeVars.GetComponent<HilfeVars>().LakeLerp.ToString());
125                     }
126                 }
127             }
128         }
129         else //Auf Feld kann nicht geklickt werden
130         {
131             FieldStatusDisplayDraw(0); //Rotes Feld erzeugen
132         }
133     }
134     else //Auf Feld wurde schon geklickt
135     {
136         if (fieldStatusDisplay.gameObject == null)
137         {
138           FieldStatusDisplayDraw(1); //Blaues Feld erzeugen
139         }
140         if (positionX < 12)
141         {
142             if (hilfeVars.GetComponent<HilfeVars>().getCount() >=  1 && gridManager.tileArray[positionX + 1, positionZ].GetComponent<IceFieldBehaviour>().GetFragile()
143             {
144                 compassDisplayDraw(0); //1+ gesammelte Äste: Kompass anzeigen, wenn Feld rechts brüchig
145             }
146         }
147         if (positionZ < 14)
148         {
149             if (hilfeVars.GetComponent<HilfeVars>().getCount() >= 2 && gridManager.tileArray[positionX, positionZ + 1].GetComponent<IceFieldBehaviour>().GetFragile())
150             {
151                 compassDisplayDraw(1);//2+ gesammelte Äste: Kompass anzeigen, wenn Feld oben brüchig
152             }
153         }
154         if (positionZ > 0)
155         {
156             if (hilfeVars.GetComponent<HilfeVars>().getCount() >= 3 && gridManager.tileArray[positionX, positionZ - 1].GetComponent<IceFieldBehaviour>().GetFragile())
```

Abb. 19 – Klasse „IceFieldBehaviour“ Teil 3

Die Methode „fieldStatusDisplayDraw()“ erzeugt je nach Parameter „int statusMode“ ein 2D-Sprite in einer bestimmten Farbe. Dieses wird knapp über dem eigentlichen klickbaren Feld platziert und nach kurzer Zeit wieder zerstört, da es nur angezeigt werden soll, wenn man mit der Maus darauf zeigt. Handelt es sich um ein blaues Feld um ein blaues wird es nicht wieder zerstört, denn der Spieler soll permanent sehen können, welche Felder er bereits angeklickt hat. (Abb. 20)

```
172         5 Verweise
            public void FieldStatusDisplayDraw(int statusMode)
173     {
174         switch (statusMode) // Modi: 0 = rot (nicht klickbar), 1 = blau (schon geklickt), 2 = grün (klickbar)
175         {
176             case 0:
177                 Debug.Log(name + " --> unclickable");
178                 fieldStatusDisplay = (GameObject)Instantiate(Resources.Load("RedCircle"));
179                 break;
180             case 1:
181                 Debug.Log(name + " --> already clicked");
182                 fieldStatusDisplay = (GameObject)Instantiate(Resources.Load("BlueCircle"));
183                 break;
184             case 2:
185                 Debug.Log(name + " --> clickable");
186                 fieldStatusDisplay = (GameObject)Instantiate(Resources.Load("GreenCircle"));
187                 break;
188         }
189         fieldStatusDisplay.name = "status_" + name;
190         fieldStatusDisplay.transform.position = transform.position + new Vector3(0, 0.1f, 0);
191         switch(statusMode)
192         {
193             case 1:
194                 break;
195             case 0:
196             case 2:
197                 Destroy(fieldStatusDisplay, 0.045f);
198                 break;
199         }
200     }
```

Abb. 20 – Klasse „IceFieldBehaviour“ Teil 4

Auf die Erklärung weiterer simplerer Klassen wurden aus Platzgründen an dieser Stelle verzichtet. Die Kernlogik des Rätsels sollte jedoch aus den eben aufgezeigten Klassen und Methoden ersichtlich geworden sein.

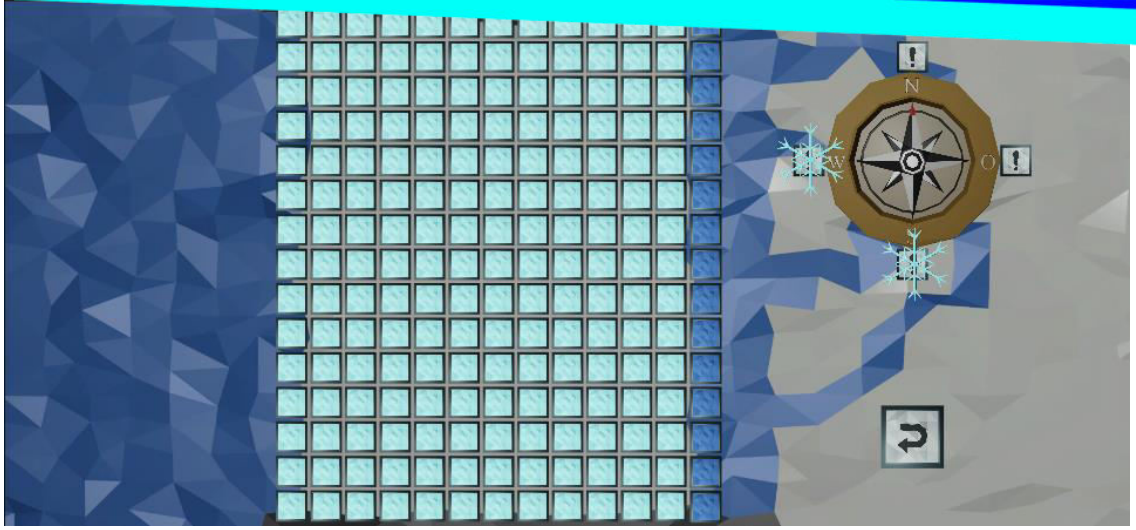


Abb. 21 – Labyrnith-Rätsel im fertigen Spiel (Unity)

3. Weitere abschließende Arbeit

Nach Fertigstellung der Haupträtsel und der wichtigsten optischen Elemente entschieden wir uns die Insel noch zu dekorieren. Hierzu modellierte ich eine Fackel, sowie Felsengruppen und schmückte die Insel anschließend mit diesen Objekte.

Abb. 22 - Fackel

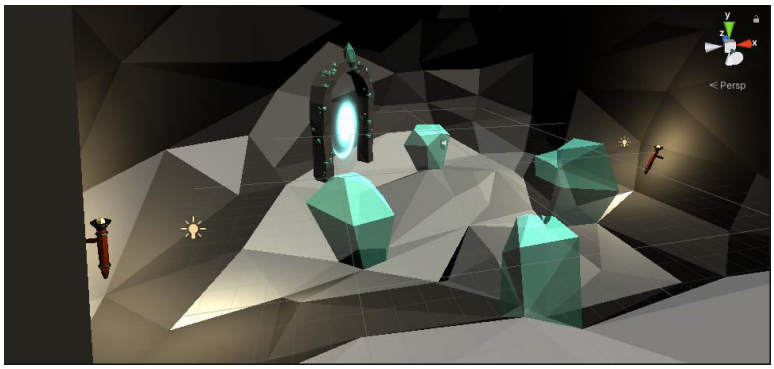


Abb. 23 – Fackeln in der Spielszene

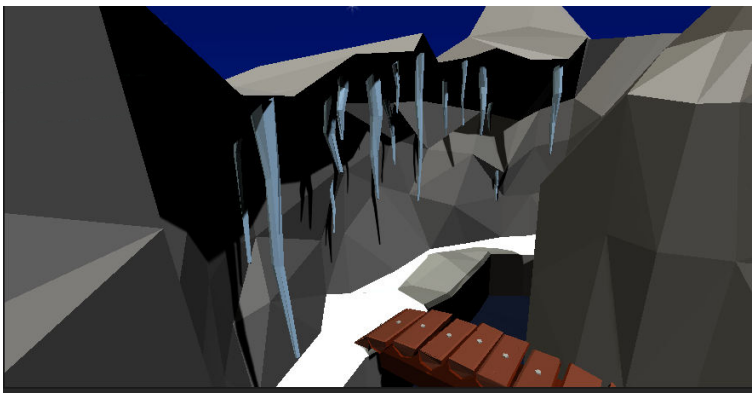


Abb. 24 – Eiszapfenvorsprung in der Spielszene

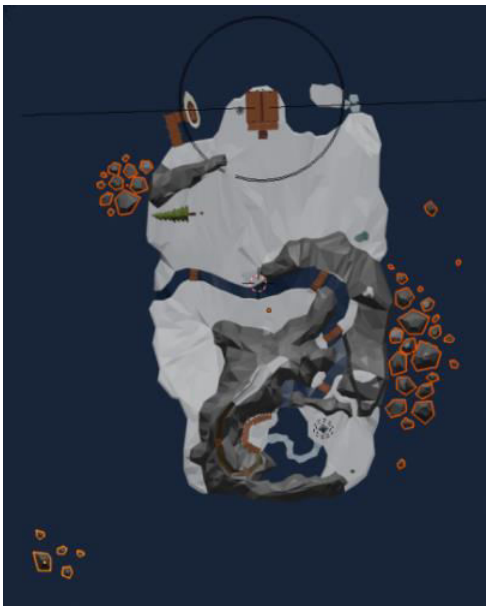


Abb. 25 – Felsengruppen in der Spielszene

4. Quellen

Modellieren Inspirationen:

https://www.google.com/search?q=low+poly+snow+scenery

https://plottkompott.de/produkt/kompass/

https://www.istockphoto.com/de/vektor/fackel-realistische-fackeln-mit-feuer-zwei-fackeln-gm959082280-261889202

https://scryfall.com/card/csp/142/thrumming-stone

https://d.wattpad.com/story_parts/1/images/16079eb1c0d658f1846801879382.jpg

Programmieren Quellen:

Udemy-Kurs von Rene Bühling

https://answers.unity.com/index.html (Fehlerbehebungen)

https://www.youtube.com/watch?v=u2_O-jQDD6s&ab_channel=LostRelicGames (Gitter erstellen)

https://www.youtube.com/watch?v=oEywwHERy1U&t=244s&ab_channel=KristerCederlund (Raycast)

https://www.youtube.com/watch?v=alPE3v9I6nY&ab_channel=HowtoGameDev (Objekte de-/aktivieren)

https://www.youtube.com/watch?v=Dl9iQ8UBi-g&t=97s&ab_channel=JasonWeimann (Lerp Funktion)

ERKLÄRUNG

Veranstaltung:
GEB5200 – GE-Lab

Betreuung:
Prof. Dr. René Bühling
rene.buehling@hs-kempten.de

Gegenstand der Erklärung

Titel der studentischen Arbeit (Projekt/Spiel):

Adventure Island – Wintergruppe

Selbstständigkeitserklärung

☒ Hiermit erkläre ich, dass ich das Projekt selbstständig bearbeitet habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken/Ressourcen als solche kenntlich gemacht bzw. angegeben habe.

Ermächtigung Veröffentlichung der studentischen Arbeit

Die Urheberin/Der Urheber der studentischen Arbeit kann (muss nicht) erklären, dass die Hochschule Kempten folgende Nutzungsrechte erhält.

☒ Hiermit ermächtige ich/wir die Hochschule Kempten zur Veröffentlichung einer Kurzzusammenfassung sowie Bilder/Screenshots und ggf. angefertigte Videos meiner studentischen Arbeit z. B. auf gedruckten Medien oder auf einer Internetseite der Hochschule Kempten zwecks Bewerbung des Bachelorstudiengangs „Game Engineering“ und des Masterstudiengangs „Game Engineering und Visual Computing“.

Dies betrifft insbesondere den Webauftritt der Hochschule Kempten inklusive der Webseite des Zentrums für Computerspiele und Simulation. Die Hochschule Kempten erhält das einfache, unentgeltliche Nutzungsrecht im Sinne der §§ 31 Abs. 2, 32 Abs. 3 Satz 3 Urheberrechtsgesetz (UrhG).

Zell, 20.01.2021 J. Dietrich Jonas Dietrich, 407414

Ort, Datum | Unterschrift Studierende/r + Name in Druckschrift, Matrikelnummer

Ort, Datum | Unterschrift Aufgabensteller/in + Name in Druckschrift