Automated generation of digital twin in virtual reality for interaction with specific nature ecosystem

Arnis Cirulis[1][0000-0001-9577-0646], Lauris Taube[1], Zintis Erics[1]

[1]Vidzeme University of Applied Sciences, Faculty of Engineering, Valmiera, Latvia
arnis.cirulis@va.lv
lauris.taube@va.lv
zintis.erics@va.lv

Abstract. This paper analyses the most suitable technological approaches to design a workflow and develop a virtual reality system - BogSim-VR, to run real time simulations for bog ecosystems (ecological systems) or peatlands in different countries and regions. Currently few technologies reflect data in an understandable way. There is also no suitable system that allows for different human actions and the visualization of consequences. Virtual reality technologies can address this problem. The BogSim-VR system is adaptable to any bog ecosystem thus creating a digital twin for experimenting with various interactions in a replicated environment.

Keywords: virtual reality, digital twin, 3D content generation, bog ecosystem, onsite simulation

1 Introduction

An already developed simulation model was the basis for the design and development of a digital bog ecosystem twin in virtual reality. Various implementations of system dynamics justify this technology [1-3]. The logic for a bog ecosystem simulation model was available in Stella Architect and Insight Maker [4-6], and is implemented in Python using calculations based on 35 criteria. The main outcome of this simulation model is generated charts, which depict the level of ground water for a specified period. This simulation model is verified and validated; it includes all components forming the hydrological system of the bog and mathematically reflects the relationships between them [4-6]. The hydrological system as an emphasis on depicting the bog ecosystem was chosen because the exact level of groundwater ensures the growth of sphagnum moss as a function of carbon storage while preventing rapid growth of forest stands, which would interfere with precipitation supplementing the water balance of the ecosystem [4-6]. Bog ecosystems have an important role in carbon sequestration and mitigation of global climate change. Bogs in the boreal and sub-arctic regions store around 15–30% of global soil carbon [7]. In the European Union, the move towards climate neutrality policy includes activities specifically aimed at the reduction of negative greenhouse gas emissions from bogs through nature conservation and renewal [8].

Bogs are one of the most endangered types of natural habitats in Europe [9] and it was important to develop the BogSim-VR system to highlight the actuality and role of bogs. This system is adaptable to any bog ecosystem thus creating a digital twin for experimenting with various interactions in a replicated environment. In a bog such interactions include cutting down trees, tree planting, digging ditches, traces of tractor equipment, fire damage etc. The BogSim-VR digital twin was developed in the Unity engine. A three-dimensional world is generated based on GIS (geographic information system) data gathered for a specific area. For a more precise three-dimensional representation LIDAR data from a drone was also used. Python simulation logic is realized in C# and currently most influence in calculations is assigned to leaf area index (LAI) values, which change depending on actions carried out in the digital twin, for example, the LAI value decreases if trees are cut down, however the ground water level increases in future predictions. The virtual environment is accessed via an Oculus Quest2 headset. Navigation uses best practices to reduce cyber sickness, and is customizable. The participant can utilize various virtual tools, instruments, devices or machinery, for example, a virtual tablet to customize interaction settings for convenient UX (user experience) and to instantly depict simulation results. By choosing a virtual chainsaw, the participant can pretend to be a forest worker and cut down trees, changing the LAI value for the area. For LAI and groundwater calculations, it is crucial to have precise data about a specific date, for example, the amount of snow and rain, temperature, sun radiation, humidity, peat depth and many other (35 variables in total). Currently data is acquired from meteorological services for the closest station in the area, but to increase accuracy of calculations, an IoT network is developed directly in the bog to gather essential parameters all year. For that purpose, narrowband Internet of Things (NB-IoT) technology is used. This allows implementing a low-power solution.
The developed virtual environment could be also used for decision makers of sensitive ecosystems or to provide a training course in environmental sciences. Since BogSim-VR system is adaptable to various bogs, it can also be customized to ecological systems, mostly land systems, like forests, grasslands and deserts. The designed flowchart and practical implementation will be used to develop similar systems, and to improve accuracy for currently developed virtual reality simulation systems.

2 Algorithms and methods to develop a digital twin and implement a simulation

The BogSim-VR system uses compute shaders from the Unity game engine [10]. The package also supplies data structures and API for programmatic access and execution. Additionally, there is I/O functionality support for reading/writing simulation data from/to CSV/TIFF files. Note that all of this is done in a manner that should facilitate reuse and extension of included components. Main features and working principles of the package include core data structures and general working principles, I/O options and formats, bog simulation specific functionality and reuse. Generally, compute shaders are programs that run on the graphics card, outside of the normal rendering pipeline. They can be used for massively parallel GPGPU (General-purpose Computing

on Graphics Processing Units) algorithms, or to accelerate parts of game rendering. To use them efficiently often an in-depth knowledge of GPU architectures and parallel algorithms is needed [10]. Various approaches are used to implement this technology in simulations, terrain visualizations, image processing [11,12] and use its benefits, especially in applications that acquire real time pipelines. There are several techniques to accelerate physics simulations [13]. Compute shaders provide support for computing various mathematical operations needed as part of the usual graphics-generating pipeline. They can also act as powerful mathematical co-processors for applications outside of the graphics pipeline [14]. General data structures used for the BogSim-VR shader include Simulation class, Frame class and Layer class. Simulation class provides the necessary API to execute simulation, move data from/to CPU/GPU and acts as a list of Frame class instances. In the middle, instances of the Frame class represent states of the simulation at every time step. This class organizes and provides mapping of Layer class instances to Layer types. At the bottom, instances of the Layer class hold the actual data of the simulation. This class provides both generic and low-level access to the data. Layer types help to identify the data held within Layer class instances as variables corresponding to the current simulation.

On the CPU side, the aforementioned classes are directly represented in C#. The Simulation class has various methods for Frame instance manipulation, compute shader execution and data transfer, it also implements IList and IDisposable interfaces for easy data access and resource management respectively. The Frame class implements the IDictionary interface for the Layer class instance and type mapping. The Layer class holds data in a byte array and provides both direct untyped access and generic unmanaged indexers in both 1D and 2D forms. Layer types are implemented and accepted as a generic enum (see fig.1.).

On the GPU side all data is stored in two memory buffers and various additional parameters are used. The input buffer holds all data from layers the simulation reads and does not write. The input parameters hold input layer dimensions. The output buffer holds data from the layers the simulation writes. This is also where the intermediate states of stocks reside. The output parameters hold output layer dimensions and 1D size for convenience as there is very little available in ways of data structuring, and extensive use of abstraction and indirection may hinder the performance, especially considering any points of synchronization during parallel execution.

The use of the developed shader package is straightforward. First, create an instance of Simulation class and add loaded Frame instances. Second, execute the simulation using the Simulate method. This method transfers input layers of the current frame to the GPU and dispatches the simulation shader, optionally calling the initialization shader beforehand. Both shaders are provided to the Simulation instance during its creation. Use the LoadLastFrame method to fetch the resulting output layers from the GPU, if necessary. Repeat the last two steps in a loop until there are no more input frames. Write the resulting frames to CSV/TIFF files.

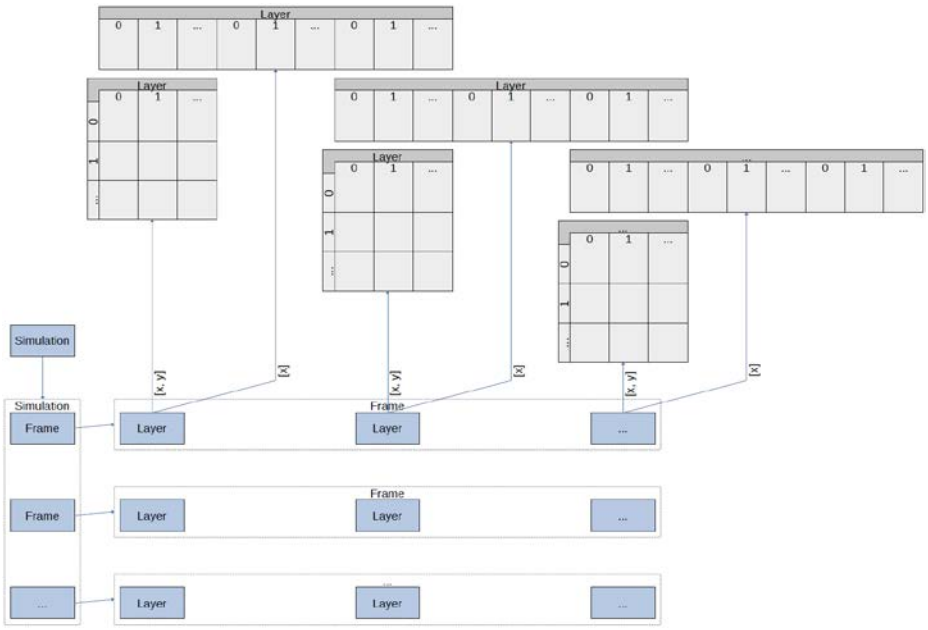

Fig.1. Simulation and class representation on CPU.

Depending on simulation and layer resolution, each frame may take up considerable memory. In this case it is anywhere from 138 to 202 MB depending on input layer resolution. While this may be acceptable for a few frames, it will reach unreasonable memory requirements for a 1000 frame simulation; therefore, two options should be considered. Providing a filter argument to the Simulation class during creation. This will enable LoadLastFrame to fetch only the layers present in the filter, and not call LoadLastFrame for every frame simulated. The SetData method used by the Simulation class to transfer data to the GPU seems to cause a memory leak if used multiple times within a single Update call, therefore refrain from simulating too many frames per single Update call (see fig. 2). The I/O facilities use a simple framework to move data in both directions. Sinks write the data to a location. Sources read the data from a location. Sinks/Sources have a limited capability to guess which specific implementation to use based on the data location identified by Uri. The simplest and least efficient supported format is CSV. Here Sinks/Sources come in three variations. As the name implies the Single variant reads/writes data from/to a single CSV file. This is also the default variant guessed when the framework is presented with a Uri pointing to a CSV file. The Grouped variant reads/writes a new CSV file for each frame of data. This variant expects a Uri pointing to a directory; therefore, it can't be guessed from Uri alone. Since the data needs to be split among multiple files, each file is named with the frame number. The Source expects data to be in this is format. The Distinct variant reads/writes a new CSV file for each layer within each frame. As with the Grouped variant, it expects a directory and cannot be guessed.

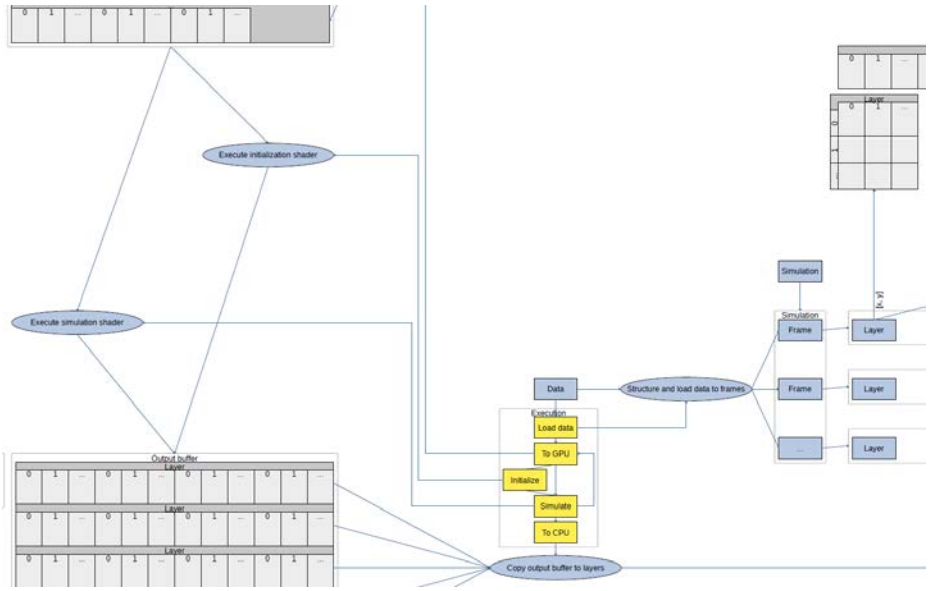

Fig.2. Fragment of general topology for the developed shader.

Again, the data is split among multiple files; therefore, each file is named and expected with the frame number and Layer type name. TIFF is the most efficient format in terms of file size and processing time. As with CSV, the Sinks/Sources come in three variations. The Single variant works with a single TIFF file and can be guessed from Uri. The Grouped variant splits data among files based on the frame and layer dimensions. This effectively places each potentially multi-spectral image in its own file because several TIFF viewers can handle multi-spectral images but cannot deal with multi-image TIFF files. The Distinct variant splits the data amongst files based on the frame and layer. This effectively places each channel in its own grayscale image. This is the most flexible format. Each file can hold multiple images and each image can have an arbitrary number of channels with an arbitrary sample width and format. As a result, this is the most efficient format but can cause some headache for viewers not prepared to handle its internal structure. This is most obvious when working with multi-image files, since most general-purpose image viewers cannot go past the first image. This format can also store georeferenced data, potentially with proper support for the time dimension. While the Sources have no problem reading such files, Sinks do not forward this information when writing as the data may come from any Source including some not storing such information.

Shader is not a traditional Source and certainly not a Sink of any kind. Instead, this Source uses an algorithm to generate data from a shader. Two algorithm variations are available: Perlin noise and Simplex noise. Both can be guessed and configured from Uri. The Scheme indicates which Source to use. The Host indicates whether to generate data in 'input', 'output' or 'all' layers. The Port indicates the divider to use to downscale the generated layer's resolution. This source is used to generate arbitrary data during development in cases where the real data is not available.

In addition to general use and data structures, there are some algorithms of note specifically for bog simulation. The terraform algorithm (see fig. 3) uses data from an arbitrary layer to change and scale a height map of the Terrain. The algorithm employs some simple checks to ensure the layer dimensions are divisible by the implied resolution. The shortest side of the layer is the resolution baseline from which the next lowest base of two is calculated. As a result, the layer used should have the longest side be a multiple of the shortest one and the shortest side should be a multiple of two. The algorithm can handle non-square layers larger than the height map but those used by Unity are square exclusively. This is compensated by scaling the terrain and should not be a problem unless extremely elongated strips of data are used, in which case an obvious visual directionality may form depending on the height map data variation. The height maps in Unity are of an odd size, power of 2 plus 1; therefore, the algorithm uses extrapolation to calculate the potential values of this semi-border. This is done purely for aesthetic reasons and has no impact on the simulation. It must be noted that there seem to be limits to the size range a height map in Unity can take, at least from the editor UI. Currently the algorithm does not account for this and some problems have been observed when using height maps below minimal resolution. The potential implications of this are unclear, but caution is advised when working outside the editor's range.

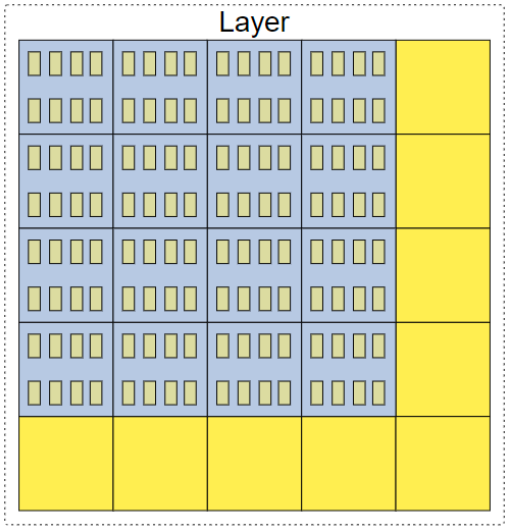

Fig.3. Terraform algorithm to change and scale the height map of Terrain.

Plant algorithm (see fig.4.) uses the data from an arbitrary layer to place various Plants. The placement of Plants is determined by their Species. These Species have several parameters directing the algorithm:

- The radius indicates the area the members of the Species affect.
- The falloff indicates how quickly, if at all, the effect of the Species wanes with distance.
- The value range indicates the limits of where the Species can spawn.

- The intensity range indicates the limits of the Species' effect on the surroundings; this is scaled from the value.
- The height range indicates the limits of the Species' physical size; this is scaled from the intensity.
- The type indicates the type of Plant the Species should spawn.

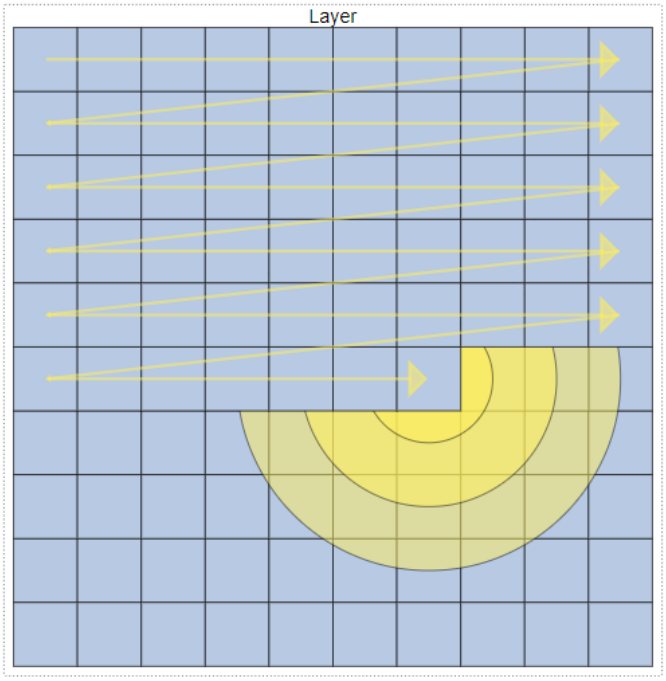

Fig.4. Plant algorithm to generate various species on Terrain.

The algorithm operates as follows; it starts at the layer origin and then moves in a row-major order along the second interpretation of the data. At each stop, it checks if any known Species can be spawned there. If a compatible Species is found, a Plant is spawned and the following surrounding area is affected. The algorithm is used to calculate potential tree positions based on the LAI (Leaf area index) layer. There is a pair of derived equations used for calculating the opposite effect on Near infrared and Red reflectance layers through the NDVI (normalized difference vegetation index) layer [5]. These change both reflectance layers when virtually planting or chopping trees. The algorithm is not based on scientific theory. It was deemed good enough at generating a canopy visually matching the distribution of the real one (see fig. 5), and the derived equations, while mathematically sound and able to maintain correct NDVI, may go into negatives in the resulting Near infrared and Red reflectance. This is mitigated by only using them in the calculation of NDVI. Still, care is advised when using the resulting reflectance values. This algorithm is rather linear and is executed on the CPU side. This is in contrast to the Terraform algorithm, which runs mostly on the GPU side.

```
# the original lai equation with embedded ndvi
lai = 3.187 * (nir - red) / (nir + red) + 0.792

# extract and invert constants
(lai - 0.792) / 3.187 = (nir - red) / (nir + red)
(0.3137747097583935 * lai) - 0.24850957012864766 = (nir - red) / (nir + red)

# assign to variables for simplicity
i = 0.3137747097583935
j = 0.24850957012864766

# add variables and move some terms
(i * lai) - j = (nir - red) / (nir + red)
((i * lai) - j) * (nir + red) = nir - red

# the original derived equations, these are recursive and kept here only as a reminder
nir = ((i * lai) - j) * (nir + red) + red
red = -(((i * lai) - j) * (nir + red) - nir)

# multiply and move some terms
i * lai * nir + i * lai * red - j * nir - j * red = nir - red
i * lai * red - j * red + red = nir + j * nir - i * lai * nir
i * lai * red + (1 - j) * red = (1 + j) * nir - i * lai * nir

# flip sides
(i * lai + 1 - j) * red = (-i * lai + 1 + j) * nir
(i * lai + 1 - j) / (-i * lai + 1 + j) = nir / red

# here's our derived equations
nir = i * lai + 1 - j
red = -i * lai + 1 + j

# those are only good for ndvi as such need to be scaled to fit reflectance ranges using reference values
k = (nirr / nir + redr / red) / 2
nir *= k
red *= k
```

Fig.5. Source code and implementation of Plant algorithm.

Since most of the code involved is abstract and generic, the reuse of this project is straightforward. Provide a different compute shader realizing the desired simulation. When providing a different Layer type enumeration matching the shader, the following needs to be considered. The enum should clearly label input and output types with 'Input' and 'Output' prefixes respectively; this determines buffer sizes. The inputs should come before the outputs; this is required since the code uses enum values as offsets in the buffers. The enum should clearly label stocks that require intermediate states with 'Stock' suffix, this is used to allocate extra memory in the output buffer. Optionally, a different value type for data interpretation should be provided.

3 Visualization of simulation data and interaction feedback delivery

To achieve the main goal a virtual reality (VR) scene should be developed for visualizing simulation data. This is done by creating a 3D environment with realistic terrain

where the user can move around, and trees to interact with. The terrain and trees are generated based on real-world data.
The user can see various types of data from the simulation in easy-to-understand ways – images and charts. The user can also interact with the trees – cut them down and see how that action has affected the simulation in real-time.
In the visualization stage the Unity project uses the Universal Render Pipeline (URP) for its graphical performance optimizations. Some custom shaders were needed for this project. For this purpose, the visual shader package 'Amplify Shader Editor' was used [15]. For VR integration 'BNG VR Interaction Framework' [16] provides movement and object and UI interactions for the user. Oculus headsets were used for development and testing, and in Unity the Oculus SDK was necessary. 'BNG Framework' provides integrations for other major VR SDKs (such as SteamVR), so other brand headsets can be used. The main logic is split into multiple scripts that perform their own specific tasks.
Simulation controller is one of the main scripts. It follows the singleton pattern and is easily accessible from any other script.
It loads data from a specific TIFF file and uses that to perform the simulation. After the simulation, its data is stored in a variable. Later this variable and its data is used in multiple other scripts. The stored simulation data includes positional data which corresponds to a position on the 3D terrain. To acquire and display various data based on position, this script also contains a method to get and return data based on that position for a specified data type.
The vegetation placer script performs the creation and placement of vegetation objects on the 3D terrain. The user can set and adjust some settings for this script in the Unity Editor. Terrain – specify which Terrain object to use. Position Offset – specify how much random positional offset to add to each tree. Vegetation Radius Multiplier – a value that adjusts how dense the forest can be (the higher the value, the less dense it is). Vegetation – a list of vegetation objects and their parameters (height, type and prefab to spawn).
For Terrain Tree Prototypes the project uses the built-in Unity terrain and tree systems. Therefore, creating and placing tree objects is not as straightforward as simply instantiating object prefabs. Before a tree object can be placed on the terrain, a specific 'TreePrototype' class object must be created. It has multiple properties, but for this use case only the 'prefab' property is changed. For each entry in the 'Vegetation' list, a tree prototype is created with the specified prefab. After that the created tree prototypes are set in terrain settings. That allows prefabs to be spawned on the terrain.
The simulation contains real terrain height data from which the 3D terrain is created. Terrain creation is done in this script, but potentially could be implemented in any other. This is done before creating and placing the tree objects. To create the terrain, the 'Terraform()' method is used from 'Simulation'. The terrain is created in the size specified in the Simulation (for example, if the Simulation is set to X = 1000 and Y = 500, then the terrain will be 1000x500 units in size on X and Z axis in Unity).
Placing the tree objects occurs as follows. Before instantiating the trees, specific 'Plant' class objects must be retrieved from the simulation. For this purpose, a 'Dictionary' is created based on values specified in the 'Vegetation' list. This is passed as a parameter

to the ‘Simulation.Plant()’ which returns a list of ‘Plant’ class objects. The ‘Plant’ object contains information about the tree’s height and position, among other parameters. Then for each of the ‘Plant’ objects a ‘Tree instance’ is created and placed on the terrain. A ‘Tree instance’ is Unity Terrain specific data and contains various parameters about the placed tree object. This data includes a prototype index to correspond to the necessary ‘TreePrototype’ entry created earlier, position according to the simulation data, random rotation on the vertical axis and random height based on specified height range in the ‘Vegetation’ list. Then the ‘Tree instance’ gets added to the terrain (see fig. 6). Two more tasks are performed while creating the instances: ‘Destroyable trees’ and ‘Overlay controller’.

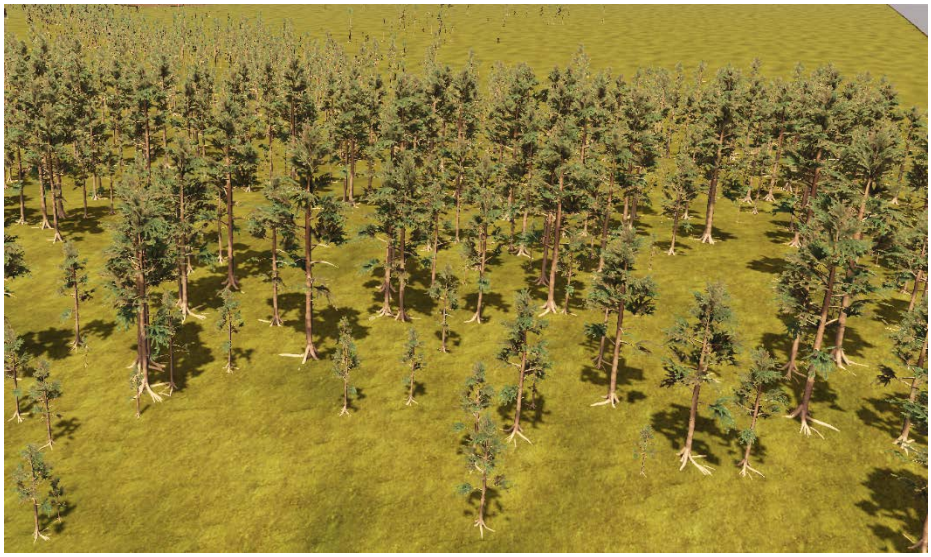

Fig.6. Generated tree on terrain.

Development of Destroyable trees (see fig.7). Unity terrain trees are not interactive by default. To be able to cut down trees, an extra object is created in the scene for each tree. When creating previously mentioned ‘TreeInstance’s, a method is called from the ‘DestroyableTreeGenerator’ script. It creates a capsule game object with a collider at the position of the tree and attaches a ‘DestroyableTree’ script component to it. The script component holds references and data of the terrain tree. The ‘DestroyableTree’ script allows the tree to be cut down. When the tree is cut down, the script removes the tree’s terrain instance and, in its place, instantiates an object prefab with the same graphics, but with an added ‘Rigidbody’ physics component that allows the tree to fall. When a tree is cut, that event is sent back to the simulation. The simulation gets refreshed and outputs new data that represents the changes caused by cutting down a tree. To achieve Tree cutter specifics, the user must use a specific object to cut down the tree: a chainsaw. The object has various components attached to it, as well as a control script for tree cutting.

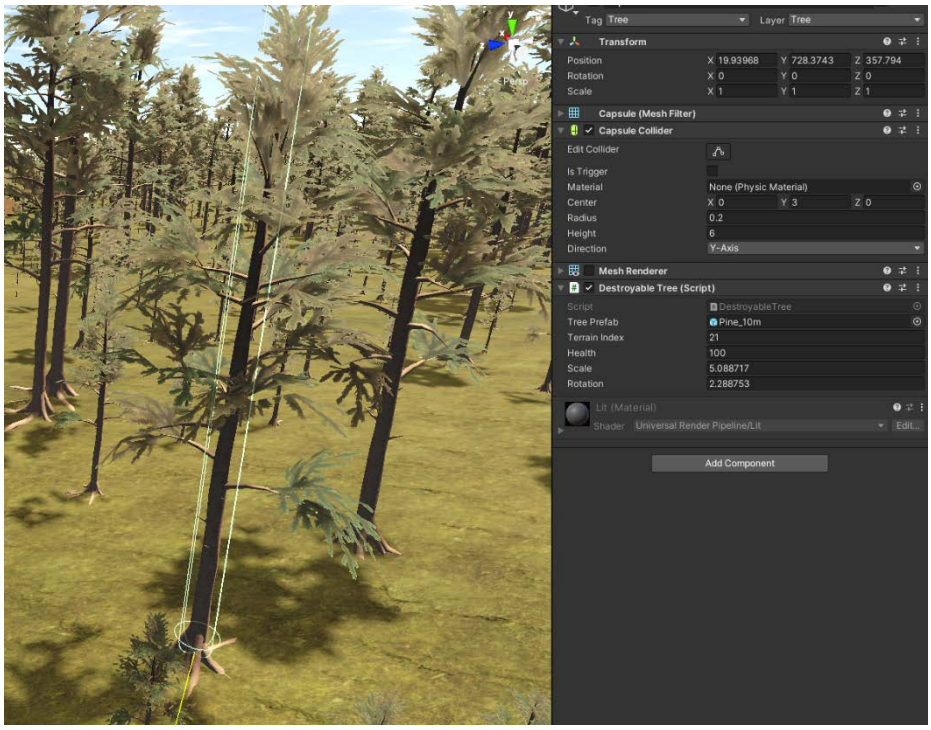

Fig.7. Destroyable tree object in 3D scene and components.

At the position of the chainsaw blade, there is a box trigger zone, which detects if a tree object is in it (see fig. 8). If there is and the user has pressed the trigger button on the controller, the cutting logic executes. Each ‘DestroyableTree’ has ‘life points’, and, while the cutting logic is being executed, they are being reduced for the tree that is in the trigger zone. If the ‘life points’ decrease to 0, the tree is cut down. While the cutting logic is being executed, a particle system and audio is played to indicate activities performed.

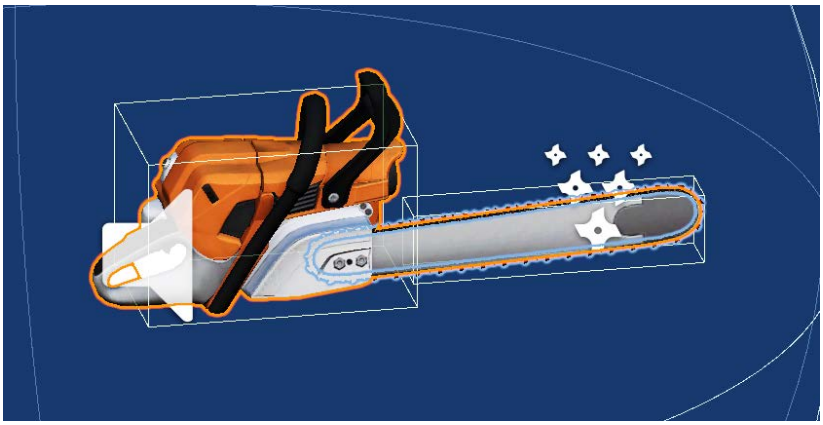

Fig.8. Chainsaw object with trigger box on blade part.

4 Discussion on interaction capabilities

A user interface (UI) is needed to display data to the user. In virtual reality a traditional screen-space UI cannot be used, it needs to be a part of the environment. For this project a tablet computer object is used, and the data and information are displayed on its screen. This is done with the Unity UI system set to work in world-space. The user has the ability to view various data in image form. Those images are displayed on the tablet object (see fig. 9). They are created and controlled by the 'OverlayController' script. These images and the data represent the 3D world and its form. It is vital for the user to know where they are in relation to the image. Therefore, a user icon is shown on top of these images to let the user know where they are. The user icon logic is described in the chapter "Overlay User Controller". When starting the application, each described image texture is created at runtime based on simulation resolution and its aspect ratio. Then they are assigned to UI references for displaying them to the user. If the image has data that corresponds to a world-space position on the 3D terrain, that position cannot be used directly to specify which pixels to affect. This position must be converted (normalized) to UV coordinates, which are in range of 0-1 on X and Y axis (2D space). The conversion is simple – the world-space position must be divided by the simulation resolution. This allows the image to be any resolution to adjust the performance and detail. This normalized value is essentially used as a positional percentage on the image – for example, a position of [0.1,0.7] on an 1000x1000 pixel image would be at [100,700]. Theoretically, if the image was the same resolution as the simulation, it could be possible to use the integer value of the world-space position directly to specify which pixel to affect. However, this approach could have a big impact on performance – larger images take longer to process.

When creating trees, their data is saved and sent to 'OverlayController'. That data contains tree position and its value. After all trees are generated, an image texture is created where each of the trees is shown as a colourful dot. The colour is determined from tree value. The position of this dot corresponds to the positions in 3D world-space.

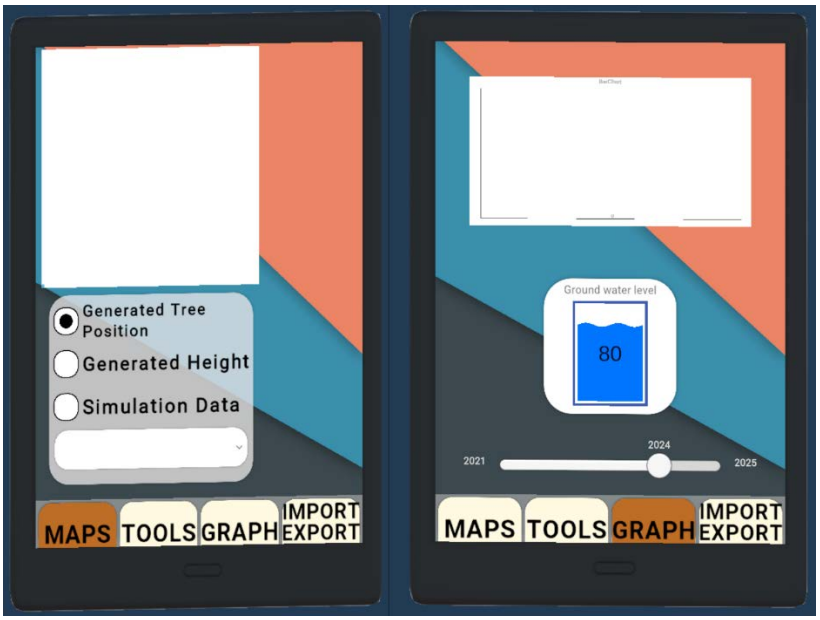

Fig.9. User interface examples and tablet object.

The resulting image texture is created by going over the list of trees, using their position to determine where each dot must be. This position is in world-space and must be normalized as described above. Dot colour is determined by using the tree value (which is in range of 0-1) in a pre-made colour gradient. That colour is then assigned to pixels in the determined position (see fig. 10). If the user cuts down a tree, the image is updated, and that trees' dot is removed (see fig. 10).

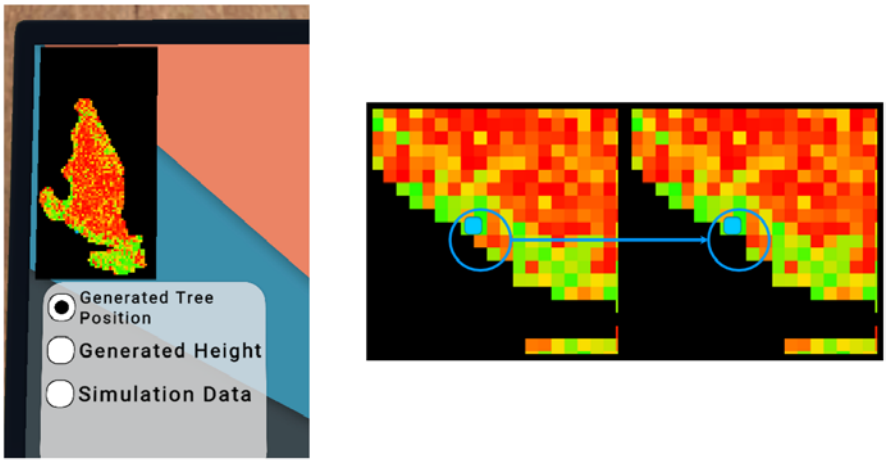

Fig.10. Generated tree representation on tree map and changes in tree map after cutting down a tree.

While simulation data already includes elevation data, we create a separate height map based on the generated terrain. This is for comparing the simulation data with what is generated, and its precision.
The height map is created by sampling the height of the terrain in a grid-like pattern. The sampling step size is adjustable and controls how detailed the resulting image will be (a lower step size means a more detailed image). Also, it is important to mention that having a lower step size impacts the speed of image generation and setting it too low will take a long time. Unity terrain has a scripting method 'SampleHeight()' that returns the height of a position. The returned value is then normalized (to a range of 0-1) to be used in a colour gradient to determine pixel colour. (See fig. 11).
The simulation has various layers with different data. To view this data in a more comprehensible way, it can be converted to images. This process is slightly different from previous image generation as layer data is in byte array form. To begin generating the image, simulation layer data needs to be converted to float values to create colour data. This is done by 'BitConverter.ToSingle()'. The returned float value is then used to create a grayscale colour using it for all three colour channels. This is done for all bytes in the array, and each created colour is added to a list for later use. Finally, the texture is created by setting pixel colours with the aforementioned colour list using 'Texture2D.SetPixels()' method.
For some image overlays it is important to know where they are in relation to the user. For this a user icon was created. It is always on top of the image overlays and follows the players' position. The logic for moving the user icon is as follows; get the user position in the 3D world and normalize it to a range of 0-1 based on simulation resolution so it can be used on any size overlay image – the normalized value is used as a positional percentage on the image. Then set the icons' position to a value that is calculated with this formula "overlay image size * normalized position".

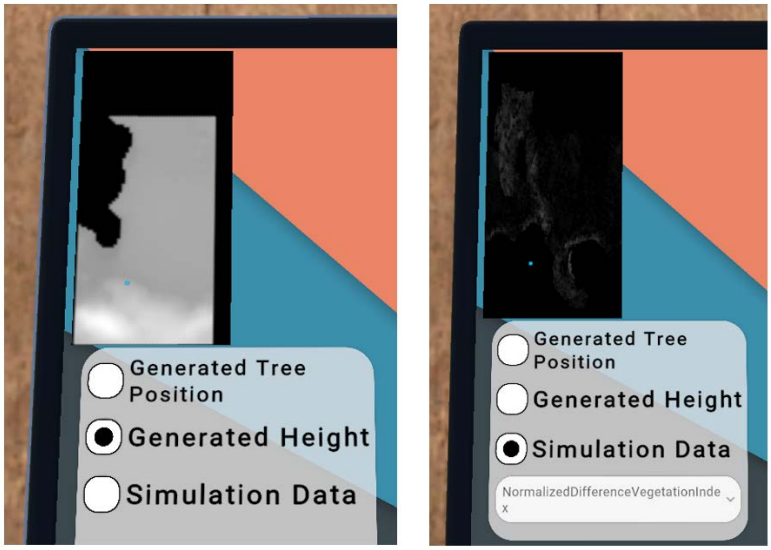

Fig.11. Generated height map and simulation data in image form.

In addition to image-based data output, simulation data can also be displayed as a chart. For the chart logic and visuals, a third-party asset was used - 'XCharts' [17]. This project uses bar charts. Data displayed is based on the users' position in 3D space and is updated in real-time as the user moves around. The logic for this is simple – each frame the game runs, the application checks if the player position has changed, and if it has, get data for the position from the simulation and send it to the chart to show that value. The 'XCharts' asset provides built-in easy updating of data– by specifying which chart an entry to update and new data values, the chart gets updated.
For this project various tree 3D models were needed. To make the creation process easier and simpler, a third-party asset was used to generate the tree models – 'MTree' [18]. It provides various useful features that Unity's built-in tree creator does not. Multiple pine tree models at different stages of growth were created with this tool. Since a large amount of tree models can cause performance issues when creating forest environments, some of the 'MTree' optimization features were used. Mainly adjusting the polygon count to be as low as possible while still maintaining a good overall look of the trees and using the automatic generation of LOD (Level of detail) objects for the model. LOD objects are copies of the original object, but with decreasing number of polygons. The LODs are useful at runtime when trying to maintain a good number of objects that the user sees while decreasing the performance as little as possible. Essentially, the tree objects that are closer to the user are shown as the higher detailed LOD objects, but the objects farther away from the user are shown as less detailed. Since it's generated at runtime, texturing by hand is not possible. For this reason, a special shader was created in 'Amplify' shader editor. This shader is based on terrain height, where at different heights a different texture is shown. The user needs to supply four textures to be used and adjust height settings. These textures must be tile-able and seamless, and to provide adequate detail, the textures need to be repeated multiple times across the terrain. By doing this, the repetition of the textures becomes very visible. To circumvent this, random noise-based colour variation was introduced – by mapping Perlin noise through a colour gradient, a seemingly random colour texture was created. This texture was then overlaid over the base textures. This broke up the repeating patterns and resulted in realistic ground texturing.

Conclusions

The BogSim-VR virtual reality system is functional and can be rated as TRL (technology readiness level) 3 - 4. The system was designed and developed to support the requirements and features of digital twin technology offering a 3D replica of a real bog ecosystem located in northern Latvia. The system prototype was tested and verified at Vidzeme University of Applied Sciences, Virtual Reality Technologies laboratory (ViA VR-Lab). The BogSim-VR system will be approbated in the next phase by performing statistical analysis and user tests. Prototype functionality improvements are

also planned. Future activities involve actions to decrease the time of digital twin generation, integration of adjustment settings and improvement of UX (user experience) for convenient interaction and cybersickness reduction.

Acknowledgements

This work is research project funded by Latvian Council of Science, project number: lzp-2020/2-0396. Project name: Visualization of real-time bog hydrological regime and simulation data in virtual reality. Research activities took place at the Faculty of Engineering at Vidzeme University of Applied Sciences, and specifically, in the Virtual Reality Technologies laboratory (ViA VR-Lab). The laboratory was established in 2009 in cooperation with the Fraunhofer Institute Virtual Reality Training and Development Centre (Magdeburg, Germany) and the University of Agder (Kristiansand, Norway), pointing to its long history and years of experience. The activities of the ViA VR-Lab include industry training, urban planning, interactive study tools and equipment in medicine, visualization solutions in logistics, tourism and history, entertaining educational environments, marketing and product demonstration.

References

1. Wu, Xiuyu, et al. "Impacts of lean construction on safety systems: a system dynamics approach." International journal of environmental research and public health 16.2 (2019): 221.
2. Muravev, Dmitri, et al. "The introduction to system dynamics approach to operational efficiency and sustainability of dry port's main parameters." Sustainability 11.8 (2019): 2413.
3. Alefari, Mudhafar, Angel Maria Fernández Barahona, and Konstantinos Salonitis. "Modelling manufacturing employees' performance based on a system dynamics approach." Procedia CIRP 72 (2018): 438-443.
4. Java, Oskars. "The Specification of Hydrological Model Requirements for Bog Restoration." Baltic Journal of Modern Computing 8.1 (2020): 164-173.
5. Java, Oskars. "Restoration of a degraded bog hydrological regime using System Dynamics modeling." CBU International Conference Proceedings. Vol. 6. ISE Research Institute, 2018.
6. Java, O.; Kohv, M.; Lõhmus,A. (2021). Performance of a Bog Hydrological System Dynamics Simulation Modeling an Ecological Restoration Context: Soomaa Case Study, Estonia. Water, 13, 2217; https://doi.org/10.3390/w13162217
7. Limpens, J., Berendse, F., Blodau, C., Canadell, J. G., Freeman, C., Holden, J., Roulet, N., Rydin, H., Schaepman-Strub, G. 2008. Peatlands and the carbon cycle: from local processes to global implications – a synthesis. Biogeosciences, Vol.5, pp.1475–1491, https://doi.org/10.5194/bg-5-1475-2008
8. IPBES (The Intergovernmental Science-Policy Platform on Biodiversity and Ecosystem Services) 2019. Kopsavilkums ricibpolitikas veidotajiem, [Summary for policy makers]. 4. lpp., A.4. (In Latvian)
9. Kiely, G., Leahy, P., McVeigh, P., Laine, A., Lewis, C., Koehler, A.K., Sottocornala, M. 2018. PeatGHG - Survey of GHG Emission and Sink Potential of Blanket Peatlands. EPA Research, pp.1-35.
10. Compute Shaders. Copyright 2016 Unity Technologies. Publication 5.3-X. Retrieved from https://docs.unity3d.com/530/Documentation/Manual/ComputeShaders.html

11. Khoury, Jad, Jonathan Dupuy, and Christophe Riccio. "Adaptive GPU Tessellation with Compute Shaders." (2018).
12. Tornai, Robert, and Péter Fürjes-Benke. "Compute Shader in Image Processing Development." (2021).
13. Mihai, Cosmin-Constantin, and Ciprian Lupu. "Using Graphics Processing Units and Compute Shaders in Real Time Multimodel Adaptive Robust Control." Electronics 10.20 (2021): 2462.
14. Vassilev, Tzvetomir. "REVIEW OF SEVERAL TECHNIQUES FOR ACCELERATING PHYSICAL SIMULATIONS ON THE GPU 3." (2020).
15. Amplify Creations, Amplify Shader Editor. Latest release date Jan 12, 2022. Retrieved from https://assetstore.unity.com/packages/tools/visual-scripting/amplify-shader-editor-68570
16. Bearded Ninja Games, VR Interaction Framework. Latest release date Oct 14, 2021. Retrieved from https://assetstore.unity.com/packages/templates/systems/vr-interaction-framework-161066
17. XCharts 2.0, unity-ugui-XCharts. Latest release date Dec 29, 2021. Retrieved from https://github.com/monitor1394/unity-ugui-Xcharts
18. Mx, Mtree - Tree Creation. Latest release date Jul 6, 2021, Retrieved from https://assetstore.unity.com/packages/tools/modeling/mtree-tree-creation-132433