Broomrocket: Open Source Text-to-3D Algorithm for 3D Object Placement

SANJA BONIC, TU Berlin, Berlin, Germany
JANOS BONIC, FernUniversität Hagen, Hagen, Germany
STEFAN SCHMID, TU Berlin, Berlin, Germany

Story writers and other creative professionals often rely on concept artists to visualize and then further iterate on their work during game development and other visualization processes. This exchange and its various stages are time-consuming, and there is no easy remedy for creating a walkable 3D concept art without involving a 3D artist yet. As a first step, we present Broomrocket, an open source text-to-3D algorithm for 3D concept art. Broomrocket's contribution is an object relation and placement algorithm that transforms user input describing a 3D scene given in plain English language into actual models placed in a 3D scene. It runs locally using an existing downloaded natural language processing model and does not require third party services unless a connection to an online 3D model distribution platform is desired. In that case, Broomrocket will search for the keywords from the user's narrative input and desired license, and place them in the 3D scene, adding each model's individual license to a license file for further usage.

CCS Concepts: • **Software and its engineering → Application specific development environments**; • **Human-centered computing** → *Visualization toolkits*; • **Applied computing → Computer games**.

Additional Key Words and Phrases: text-to-3d, level design, scene generation, 3D concept art, prototyping, language processing, computing

1 INTRODUCTION

For the creation of games, we are on a quest to produce content faster and better. In this process, we need a mix of skill sets covering various roles, including but not limited to developers, artists, and writers. Turning text into concept art usually requires multiple people and several iterations, which leads to inefficiencies when validating initial ideas. Writers who lack the experience in 3D software do not currently have an easy way of showcasing their vision of a world without involving one or more artists. Creating 2D concept art has recently been expedited through machine learning algorithms, while 3D scene generation is not yet widely available nor usable.

Looking at the historical timeline of 3D scene generation, we see that there are many papers discussing a few use cases, mainly for educational projects, e.g. [1, 2, 13, 15, 16, 27]. The interest in 3D has risen rapidly over the last three years with many projects building upon each other. These quick iterations mean that for consumer-grade products, especially those that require real-time rendering, we cannot yet rely on much of the current research. Works such as Point-E [18] and Shap-E [12] by OpenAI, Google's DreamFusion [19], and Prolificdreamer [24] continue to develop improved machine learning models in order to generate text-to-3D objects.

One of the many interesting and innovative projects in the area of 3D scene generation originates from the Max Planck Institute [26]. It revolves around a given human animation sequence, where they use the bounding

Authors' addresses: Sanja Bonic, TU Berlin, Einsteinufer 17, Berlin, Germany, 10587; Janos Bonic, FernUniversität Hagen, Universitätsstraße 47, Hagen, Germany, 58097; Stefan Schmid, TU Berlin, Einsteinufer 17, Berlin, Germany, 10587.

ACM 2832-5516/2024/2-ART
https://doi.org/10.1145/3648233

boxes inferred from the motion paths as well as the foot vertex information to determine required free space as well as possible shapes of the room type that should be generated. With the right datasets and integrations, this could become a very valuable addition to 3D content creation.

In contrast to existing research, our work focuses mainly on formal, defined methods for generating 3D scenes by placing existing meshes, and developing better tools that can be used right now to make games and other 3D content quickly and efficiently. While there are several papers suggesting algorithms for the recognition of spatial terminology, they only recognize the spatial entities themselves, such as the work on spatial role labeling by Mazalov et al. [17]. Newer research by Hwang et al. [10] investigating ChatGPT spatial understanding confirms that machine learning tools using large language models are able to interpret spatial terminology, but we are still far away from this information being reliably used in today's tooling. Notably, Hwang et al. state that the use of ChatGPT might be valuable in order to generate drawings from textual descriptions, still addressing only 2D space. Without axis assignments, coordinates, bounding boxes, and object relations, we are unable to use the spatial terminology to assign further semantic meaning in 3D space and create scenes from it.

Tools for the procedural generation of terrains and the placement of assets are widely available in the industry with products such as World Machine [25], GeNa Pro [20], or Dash [9]. Unfortunately, these tools are either bound to a specific game engine, need additional customization of assets before usage, or only work if the scene contains a landscape. Additionally, they require a deeper knowledge of the underlying 3D software and are typically not usable without prior experience.

Our contribution to the scientific and game development community is Broomrocket, a text-to-3D object relation and placement algorithm as described and then transformed into 3D space from narrative user input in plain English language. In the early stages of game development, 3D scenes and concept art often need to be discarded and redesigned through many tedious iterations until the look and feel fits the vision for the game. The main goal of this algorithm is to allow game writers, game designers, artists, and others to quickly iterate on concept art, lore, and scene ideas in order to speed up the game development process.

Since we want to keep Broomrocket lightweight, reasonably fast, and specific to 3D scene creation, we use a general language model and develop our own formal methods on top of it, in order to keep the algorithm flexible and portable. Our goal is to combine spatial terminology including object references with actual semantic meaning through a placement algorithm in 3D space.

We are not aware of any similar algorithm and implementation that allows for general purpose 3D scene creation from plain text, which is why Broomrocket is a novel contribution building upon work in the NLP field and existing 3D tools. In order to showcase Broomrocket's portable algorithm, we developed an open source MIT-licensed Python add-on for Blender [4] as well as an implementation for the open source game engine Godot [8]. Broomrocket can run locally and without third party services by using either a default cube or an asset folder of your choice. In case an asset store is desired, we also include a connection to the online 3D model distribution platform Sketchfab [21], where the algorithm uses keywords from your sentences to search for 3D models using your license preference. The large majority of our provided Python code [22] is not specific to Blender and can be ported to other Python-supporting 3D software by implementing the abstract classes *Engine* and *LoadableMesh*. While the Blender add-on can run from within Blender, we also provide a separate engine-independent network server, which can either be hosted or run locally to provide the NLP, placement, and object relation functionality. This should allow for better integration points for any 3D engine. One such integration showcase using the server is our Godot implementation.

In this paper, we first look at the motivation and contribution of Broomrocket (section 2) before moving on to the overview of Broomrocket's components (section 3), implementation (section 4), results (section 5), evaluation (section 6), and conclusion (section 7).

2 MOTIVATION

During the development of our first commercial game, we attempted to follow the advice echoed throughout the industry of creating a Steam store page for the game as early as possible. With different parts of the game still in various stages of readiness, we needed screenshots using our existing assets that would reflect the final gameplay. As we were looking for the ideal parts of the game to present, we would create and recreate the scenes countless times, assembling and resizing them by hand each time. Reflecting on the process of creating the store page, we decided to look for tools that could expedite the creation of these screenshots in the future.

Procedural generation of levels is wide-spread in the industry. Tools such as World Machine [25], Gaia Pro / GeNa Pro [20], Dash [9] or Unreal Engine's own procedural generation features [5] provide the ability to speed up the process of creating terrain, roads, rivers, or spawn assets. Their capabilities and fitness for early 3D concept art work vary, but a common trait among all of them is the need for experience in working with 3D software or the particular game engine they build upon. As we do not aim to provide a complete state of the art survey of the currently available procedural generation tools, we describe a select few of them in relation to Broomrocket in the following table 1.

Name	Editor	License	3D/Game engine
World Machine	Mouse-based	Proprietary (free for non-commercial use)	Standalone software
Gaia Pro / GeNa Pro	Mouse-based	Proprietary	Unity
Dash	Hybrid	Proprietary	Unreal Engine
Unreal Engine Procedural Content Generation	Visual programming	Proprietary	Unreal Engine
BlenderGPT	Text-based	MIT (requires ChatGPT subscription)	Blender
Broomrocket	Text-based	MIT	Blender, Godot (portable)

Table 1. Procedural/assisted scene building tools

While all of these tools have a solid use case, they do not support workflows for people who primarily work with text. Text-based editing tools such as BlenderGPT [7] are available, but they work by prompting ChatGPT or other generative AI code assistance tools to generate and run code directly on the user's computer which can cause security and compliance issues [11].

With Broomrocket, we contribute a robust, extensible object relation and placement algorithm that can be used and built upon today. Broomrocket supports the creation of 3D environment scenes from textual descriptions to mesh components in a 3D scene quickly and efficiently. In order to achieve this, we use existing meshes and a self-defined corpus of keywords that can be mapped to from plain English text. The main feature that Broomrocket delivers compared to other text-based tools, such as Dash or BlenderGPT, is reproducibility by allowing the same series of sentences to produce an exact replica of the scene with different assets, depending on the chosen asset library. We hope that it is useful for the wider game development and 3D creator community as well as for further research in this area.

3 OVERVIEW

Broomrocket consists of three components that make it useful and extensible: Natural Language Processing (NLP), an object relation and placement algorithm as well as a mesh provider plugin system.

3.1 NLP

Early in the development phase, we made a decision to use NLP as we did not want people to have to learn yet another markup language or additional syntax. Everyone should be able to quickly iterate on scene design ideas, even with little to no 3D modeling or programming background. While game development is not the only application for Broomrocket and the general concept behind it, it is the main use case and target audience for us. We could have chosen to use a JSON-style or similar syntax to better and easier represent a scene in terms of coordinates, sizing, and more, at which point creating this data representation would mean that we would have had to reinvent the wheel of VRML [23], or any of the many other proprietary and open source 3D model formats which would have yielded no obvious benefit or novel applicability. Hence, we decided to make a tool and respective algorithms that work with the existing tools and skill sets of content creators. For that purpose, NLP seems to be the most appropriate choice.

Our main logic loop for the NLP part, for which we used SpaCy 3.5.3 [6] and one of its core small language models, *en_core_web_sm 3.5.0*, is to extract the relevant nouns, prepositions, and their relation to each other. One advantage of SpaCy is that it comes with a dependency parser whereas libraries like NLTK [3] call out to a Java implementation to achieve dependency parsing or need many additional dependencies and installations. For our use case of quick prototyping with editing capabilities for final touches, we needed to make it as easy as possible for people to install and use while running the software locally. This requirement is met by SpaCy.

In its current implementation, Broomrocket allows for sequential text field input as well as file input for reproducible idea validation. For a quick test, adding input to the text field is sufficient whereas for iterations on a 3D scene, the file input can be very useful. After the input is processed and we understand the relations of recognized entities to each other, we start loading the downloaded models and placing them inside the scene.

3.2 Object Relation and Placement Algorithm

Each sentence is processed and its models placed individually, but one feature of Broomrocket is that it knows what is in the scene, which means that the user can refer to something that has already been placed by the initial reference name given. For example, in the sentence sequence *'Make a house and a garden.'*, *'There is a forest behind the garden.'*, and *'Add 2 flower pots in front of the house.'*, our algorithm can recognize the number of flower pots as long as it is given in its numerical form, place the pots in front of the previously placed house, and will correctly add a forest behind the previously placed garden. Since the garden has no prepositional attachment, it is distributed on the assigned axis alongside the house. We decided not to add additional dependencies to Broomrocket which is why spelled out numbers only work up to *ninety-nine thousand* with this version of Broomrocket and we do not support constructs such as *dozens, ounces, hundreds of thousands, millions, etc.*.

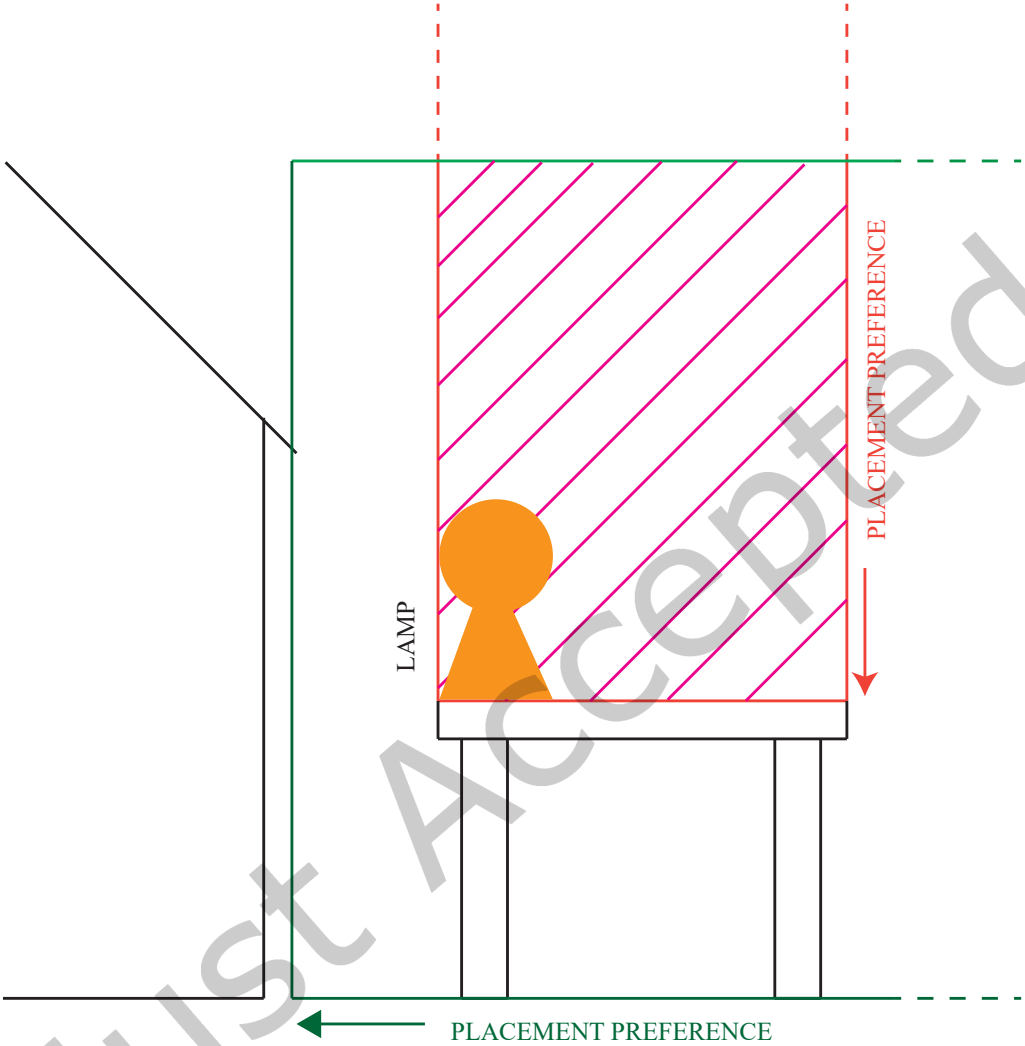
When we place objects in the scene, we give the objects the name that was queried, meaning that an initial sentence like *'Make a house.'* will result in a house object with the name *house* added to the empty scene. This allows us to query the placed object hierarchy and position follow-up objects next to the object reference for an object that was mentioned at the start of the scene creation.

Once the objects to be added are clear and we have checked for existing objects, we move on to the spatial representations and the models' coordinates. In its initial version, Broomrocket supports the following features:

- We added representations of *mm, cm, m, feet, inches* and various other units to Broomrocket so that these instructions can be properly recognized and used for placement with the x, y, and z axes in relation to the object.
- Commonly used prepositions, such as *left, right, next to, up, down, above, under, below, in front of, behind, between, at the back/front, in the background*, are converted to their respective axes and placed accordingly.

One limitation of this system is that we cannot rely on meshes having a semantically correct orientation. For this reason, we designed Broomrocket with the following internal coordinate understanding: front is $+x$, back is

Fig. 1. Visualization of Broomrocket's internal coordinate understanding and bounding box representation



Behind the house, there is a lamp on the table.

$-x$, left is $-y$, right is $+y$, top is $+z$, and bottom is $-z$. Each individual direction is calculated separately, based on the new object's reference object, and results in a bounding box as well as a preferred location within that box, as shown in Fig. 1.

The final placement of a mesh is calculated as an intersection of all directional bounding boxes as well as the first location mentioned in the sentence. For the example sentence of *'Behind the house, there is a lamp on the table.'*, that means that there are two directional bounding boxes representing the reference objects for our new object. One directional bounding box, the house, stretches from its furthest point on the $-x$ axis to infinity. The

other directional bounding box goes from the top of the table, which is represented by the highest z coordinate, to $+z$ infinity. As the first mentioned location is *behind the house*, Broomrocket takes that as the origin placement. This results in the lamp being placed at the edge of the table, rather than in the middle, since we do not yet have a dataset with such high level semantic labelling that we could infer that a lamp would rather be in the center of the table.

3.3 Mesh Providers

Broomrocket allows for extensibility with regards to datasets through its plugin system which currently features three mesh providers: cube, custom folder, and Sketchfab. Further mesh providers could be implemented using any other 3D model distribution platform or dataset. Our cube mesh provider uses a 1x1m cube as default shape for any object that needs to be placed. It is the default fallback in case a keyword cannot be found using other mesh providers. For the online 3D model distribution platform showcase, we chose Sketchfab [21] for creators who aim at getting more of a look and feel for the game they are developing than default shapes can achieve. In addition, users may also pick a folder with their own assets, where our algorithm will look for the given keywords and either place the requested model or the default cube if the keyword cannot be found inside the folder.

Currently available 3D datasets used in research usually either cannot be used commercially, are raw photogrammetry scans, or contain only a very small niche category of models and poses. With this limitation in mind and our initial prerequisite of making Broomrocket available for use for everyone, we decided to create a usable mix of these three mesh providers allowing 3D scene generation entirely from a default cube, a custom asset library, or user-choice licensed models found on Sketchfab which can either fit the content's needs or be replaced once the prototype level or game design are found to be satisfying. APIs are usually rate-limited, though, as is also the case with Sketchfab, which is why we recommend creating your own asset library with curated or created models. Broomrocket is meant to be lightweight, fast, and allow for quick iterations in the early stages of 3D content creation.

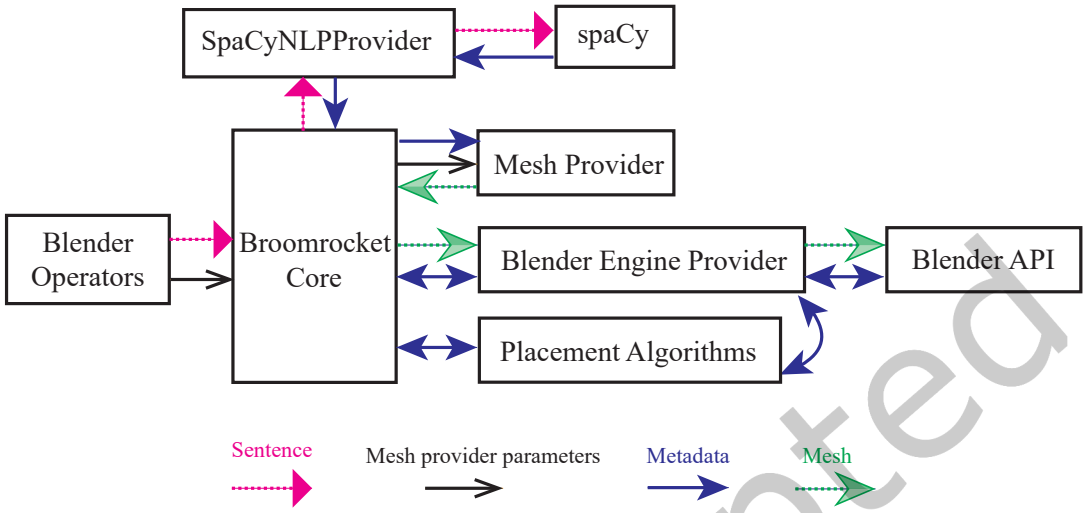
In order to satisfy licensing requirements by licenses from Sketchfab that require attribution, we also generate a license file with the created 3D scene that should be included in the release if models are kept and not replaced in the final version of the game or other 3D content that is created. Since Broomrocket's implementation is open source, everyone can add additional datasets and search parameters to fit their needs and licensing requirements, allowing for high extensibility and additional application areas.

4 IMPLEMENTATION

As a static 3D scene generation algorithm using natural language user input, Broomrocket is still fairly limited in its uses. In this section, we describe the current implementation steps as additional information to the commented code that is made available as open source under the MIT license. For the concrete implementations, we chose Blender as it is widely used in the games and 3D content creation industry by both independent artists as well as established studios, and the open source game engine Godot. Due to their open source nature, they lend themselves well to academia and are very extensible. We chose not to restrict our prototype to creating a minimalistic rendering engine with scene exports as we want others to be able to use our code and algorithms as well as the outputs they create to extend and edit them even further. With Blender and Godot as the 3D tools of our choice, we used the relevant APIs to create easily usable add-ons that could be installed from the respective interfaces if someone wanted to maintain the releases. For the purposes of this paper, we only provide installation instructions and the source code itself [22].

Once Broomrocket is loaded, the user can provide their input either via a text field, or load multiple sentences from a file. In either case, the input is segmented by punctuation and is recorded in the interface as individual items. The user has the choice to run only selected sentences from the list or run all of them at once.

Fig. 2. High-level architecture diagram of the internal components and data flow of Broomrocket for Blender



Internally, Broomrocket processes sentences as separate entities and completely finishes processing one sentence before it starts the next one. This allows Broomrocket to partially process input in case a sentence fails, and it is necessary for correct object relation since sentences may refer to previously placed meshes. The high-level architecture diagram in Fig. 2 shows the internal components and data flow of the example implementation for Broomrocket's algorithm.

4.1 NLP

As a first step, a sentence is processed into tokens by SpaCy. We use these tokens to construct a placement data structure. Initially, we tried to use the tree structure created by SpaCy, but soon discovered edge cases where the tree structure is incorrectly linking conjunctions or incorrectly recognizing entities which led to faulty results, as further explained in the results (Section 5).

Instead, we had to resort to implementing a custom algorithm that links nouns in a sentence to either a verb or an adposition. The verb-linked nouns provide information about the objects the user wants to add to their scene, while adposition-linked nouns are a first clue to the prepositions we need to recognize spatial terms that may be translated to 3D coordinates. We iterate over the tokens and process nouns according to the following rules:

(1) Nouns are stored in their singular form, their plural form is marked in the final data structure.
(2) If no verb or adposition has been found yet, the nouns are added to a temporary buffer which is added to the verb or adposition when found.
(3) If a verb is found, the verb is recorded as the action to take. Previously buffered nouns and any subsequent nouns are linked to the verb.
(4) If an adposition is found, the adposition is recorded as a spatial reference word. Previously buffered nouns and subsequent nouns are linked to the adpositions.
(5) If two consecutive nouns are found without a conjunction, but with a compound marker, the noun is merged with the previous noun, delimited with a whitespace (e.g. *flower* and *pot* are recognized as *flower pot*).

(6) If two consecutive nouns are found without a conjunction and no compound marker, the linking to any previous verbs or adpositions is cleared and the following nouns are collected into the temporary buffer again.
(7) If a number is found in written form, it is translated into its numeric form.
(8) If a number is found, the subsequent noun with an entity type of QUANTITY is treated as a distance denomination. The nouns are translated into numeric multipliers of units.
(9) If a number is found and the subsequent noun is attached to a verb, the number is treated as a quantity. In its first version, Broomrocket does not support adposition-linked numerals (e.g. *behind the 2 houses*).
(10) If an adverb is found, it is recorded to be added to the next adposition (e.g. *next to the house* where *next* is an adverb and *to* is the adposition).

During various iterations of this algorithm, we found the following special cases which we fixed with the above steps in the algorithm:

- In case of the adpositions *in*, *at*, or *to* followed by a noun and then an *of*, such as *in front of*, we record the noun as the spatial reference word instead of the adposition because *front*, *back*, and some other spatial terms are recognized as nouns.
- If a noun is (incorrectly) identified as a compound noun, but the next token is *of*, we treat the noun as a spatial reference word. Such is the case with *There is a swing right of the house.*, where *right* is recognized as a noun and the *swing* is its compound, which would result in a keyword search for *swing right*.
- The imperative verb *place* is sometimes incorrectly identified a noun by SpaCy. We transform it to a verb if encountered at the beginning of a sentence.

We process the sentence into a data structure that records the action to take, a list of objects to place, and any spatial references found. Actions are simple records of the verb, which are currently not interpreted, but may in future be used for actions beyond placing meshes, such as changing materials, scaling, rotating, editing, or removing objects. The objects to place are recorded in a singular form, for search in a mesh database, along with their cardinality. Spatial references store their term, such as *behind*, along with their referenced objects, and optional distance denomination (numbered distance and units, e.g. *50 m* or *ten inches*). Referenced objects are stored in a similar fashion as the objects to place in their singular form with a plurality flag.

4.2 Object Relation and Placement Algorithm

Once the NLP processing is complete, any spatial reference objects in the scene are resolved to meshes, and their coordinates are made available. All mesh coordinates are treated according to their bounding boxes. In its first version, Broomrocket does not support convex mesh collisions.

The placement algorithm implements a plugin system where each spatial reference word can be processed by separate plugins. Each plugin must return a three-dimensional volume in which to place the mesh. If multiple spatial reference words are found in a sentence, such as *behind the house on the porch*, the appropriate plugin is called for each of these words. The resulting volumes are then intersected to produce a final volume in which the meshes can be placed. Additionally, each plugin must return the best possible position for each object that is to be placed.

Currently, we implement one plugin in multiple variations to provide support for keywords such as *behind*, *left of*, *in front of*, *above* and more. Since these are identical in nature, we describe the behavior of the algorithm using the example of the *behind* placement strategy.

The algorithm treats the negative X coordinates as *behind* an object. In future iterations, the algorithm may be improved to provide contextual placement based on the user's current viewport or object custom data, if present. The code then iterates over all reference objects paired with the current *behind* spatial reference word. Their lowest X coordinate is taken and any possible distance denomination is subtracted from the coordinate.

This value represents the upper bound of the X coordinate in the returned value, while the lower bound will be negative infinity. The Y and Z bounding box coordinates will match the bounding box of the reference object. If the objects to be placed exceed the placement bounding box, the algorithm will extend it in the Y and Z directions to fit the objects.

For each object to place, the algorithm will then compute the best spawn point translation such that the lowest Z coordinate matches the lowest Z coordinate of the reference object or objects, while the ideal X coordinate will be computed to be as high as possible. The Y coordinate will be computed such that the objects are placed in a row along the Y axis, distributed between the highest and lowest Y coordinate of the reference objects.

Finally, when all plugins have supplied their placement volumes and ideal spawn points, the volumes are intersected from each plugin and the meshes are placed. First adpositions in the sentence take placement precedence as the translations of the meshes are calculated starting with the first plugin and iterating from there. If this calculation order cannot be fulfilled due to the bounding box of further adpositions not allowing for it, for example in case of height limitations between reference objects, the next translation preference in the order is taken into account.

As previously mentioned, the above algorithm can be applied to all directional keywords, if we replace the axis and orientation accordingly.

5 RESULTS

Broomrocket's Python implementation for Blender covers over 3,300 lines of code and uses SpaCy with its small *en_core_web_sm* model, its dependencies as well as some additional dependencies for HTTP requests in order to connect to Sketchfab. For Sketchfab, users will need an account and API key which they may add via the add-on interface panel. Without Sketchfab, the code can run on users' local computers offline, using either our default cube mesh provider or the users' selected asset library folder.

In the Godot integration showcase, we only support a minimal set of features to demonstrate the algorithm itself, without implementing the user interface for inserting multiple sentences or a Sketchfab integration. The separation of functionality into an engine-independent server covers just under 3,000 lines of code, of which most is identical to the Blender add-on, representing the core of our algorithm with its abstractions. Godot-specific code comprises roughly 2,000 lines of additional code written in GDScript, including a conversion of Broomrocket's internal coordinate system to Godot's different coordinate understanding.

One of the major challenges for Broomrocket was the difference in quality of downloaded meshes that can be found online as well as their metadata. Some models are uploaded with inconsistent origin points and scaling. Broomrocket cannot account for scaling as it does not understand the semantic probabilities of a house that is 1 km long versus a house that is 10 cm long. What we did account for, though, are initial offsets, which we ignore by looking at the bounding boxes of the loaded mesh instead of taking the original model's coordinates.

Aside from individual mesh download issues, we also experienced several drawbacks to using NLP instead of clearly defined keywords, namely that we need to rely on the NLP library coming back with meaningful results, which in our very few use cases already proved to be an expected mistake. We showcase two examples (Fig. 3 and Fig. 4) where we had to build our algorithm specifically around these linguistic bugs, that are present in both NLTK and SpaCy.

In Fig. 3, any human would clearly understand the imperative form of the verb *to place* at the beginning of the sentence, but both of the NLP libraries that we tried recognize the noun *place* instead. Our workaround for this particular word was a not very elegant but workable conditional clause in our code.

A larger problem arises with the sentence *There is a pool and a ball behind the house and the garden.* (Fig. 4). The ball and the garden are recognized as conjuncted here. We cannot tell why, but assume that the determinants are not given enough weight in this particular form of sentence, leading to the sentence being recognized as

Fig. 3. Both SpaCy and NLTK recognize the imperative verb 'place' as a noun, which we had to fix for our implementation. This visualization was done using SpaCy's visualization tool.

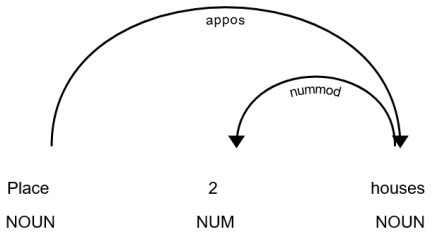
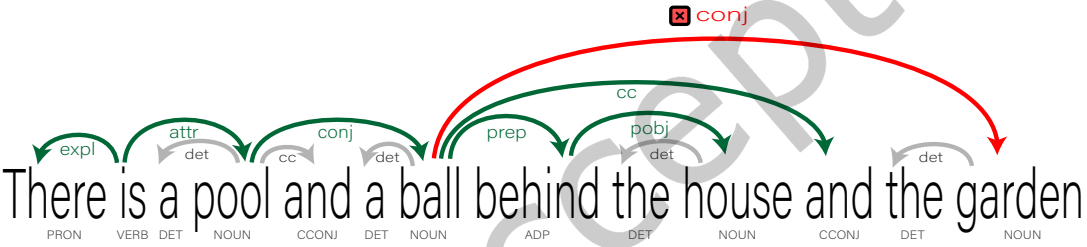


Fig. 4. A larger sentence with SpaCy's interpretation, depicting a wrong conjunction dependency between the ball and the garden.



There is a pool and a ball behind the house, and a garden. If the Oxford comma was used consistently across the English language, such a mistake might not have happened. Equally, we assume that determinants are used interchangeably in many NLP models or are given varying weights leading to *'a'* and *'the'* often being treated equally when they should not be.

Broomrocket does not account for prepositions that are not coded into the currently existing placement strategies. One possible way to fix this is to implement and test more placement strategies to cover more spatial terms, or to create scenes with descriptions that are then used to train a machine learning model. Both of these would be viable future research projects. Currently unsupported spatial terms include anything implying a collision of meshes, such as *into, inside, surrounding, etc.* as well as direct diagonal terminology, although Broomrocket does support more than one spatial term within one sentence (e.g. *behind the house on the table*), leading to more flexibility than just regular one-dimensional placement on the x, y, or z axis.

Despite its complex calculations, Broomrocket is still a 'dumb' algorithm with many workarounds in its implementation accounting for various NLP, tool-related, and mesh-specific quirks. In an ideal world where all assets were designed to be game-ready, the mesh metadata would include better positioning, labelled mesh hierarchies, orientation metadata, material slot names, and named attachment locations in order to enable a truly sophisticated object relation and placement algorithm. We show the difference in quality by using Sketchfab as the mesh provider in Fig. 5, where meshes have no custom metadata applied and the scaling of the separate objects cannot be properly adjusted. The algorithm works, but the resulting scene is not as usable for good level prototyping. In Fig. 6, we use a custom asset library, where we have created three objects – a ground, a house, and a porch. The house includes several connected vertices without a face. These faceless objects are used as

Fig. 5. Sketchfab mesh provider with licensing information for the keywords used. Licenses can be chosen by preference for the online mesh provider, allowing for immediate commercial applicability. Unfortunately, Broomrocket cannot adjust the sizing of downloaded meshes.

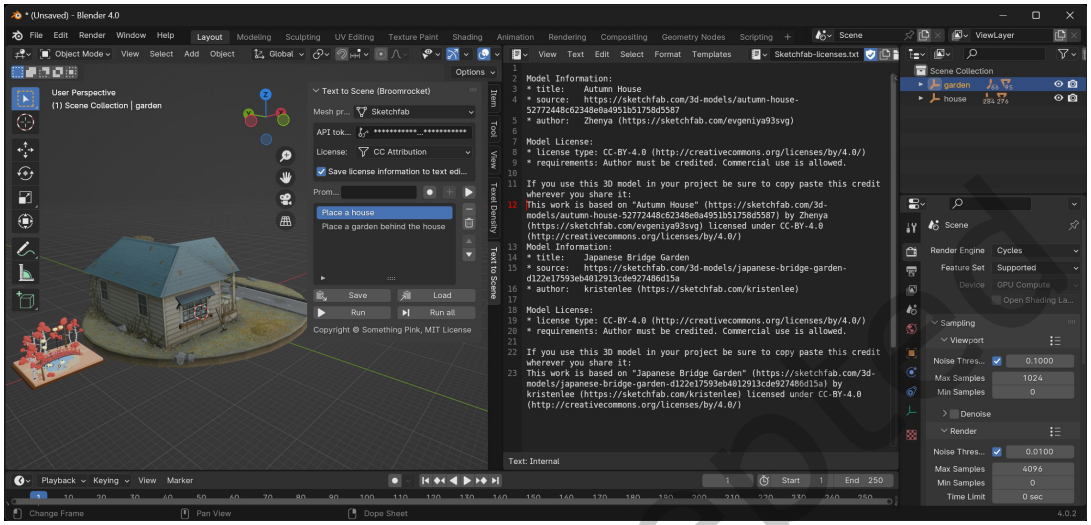
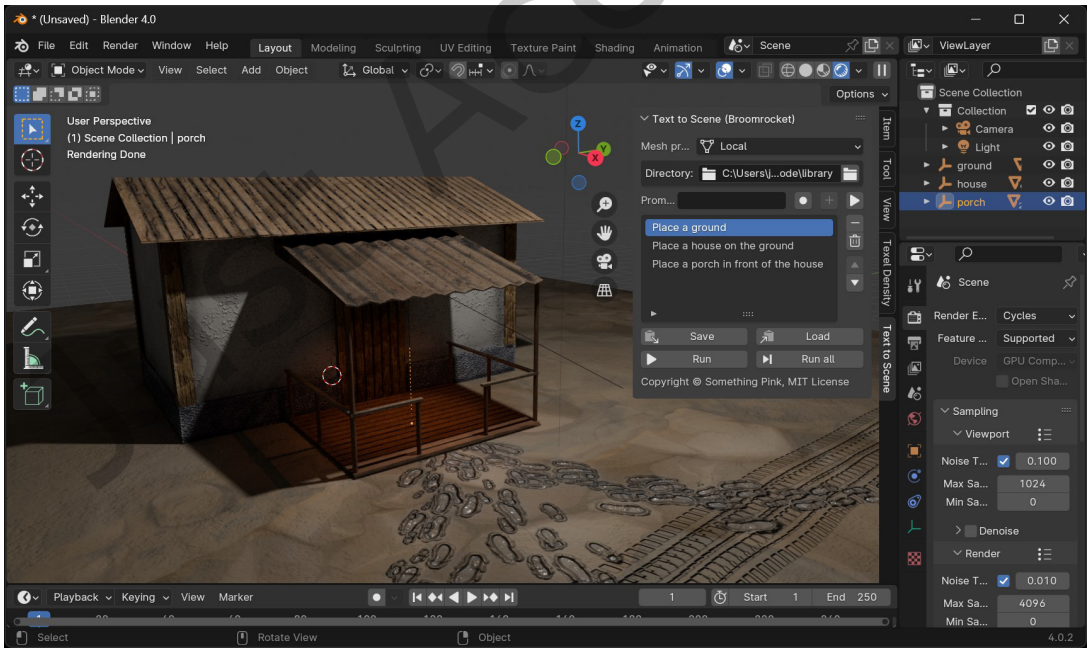


Fig. 6. Improved placement with a custom asset library that includes additional positioning metadata. The porch is located in front of the house, ignoring the overhanging roof for the *front* bounding box placement.



metadata for our algorithm to determine the correct positioning for objects, resulting in a smooth placement of the porch despite the overhanging roof in our example.

Finally, the first Broomrocket implementations do not cover common Blender or Godot commands like removing, scaling, rotating, editing, sculpting, painting, or animation. They are focused primarily on the placement and static 3D scene generation using plain English text input by both technical and non-technical users.

6 EVALUATION

We evaluated the correctness of Broomrocket's behavior through reproducible unit tests. These cover the SpaCy integration and the placement strategies and helped us find the bugs in the NLP library as mentioned in the results (section 5). The aforementioned unit tests are our primary method of formal evaluation and passed fully and reproducibly, once we fixed the bugs originating in the NLP library.

6.1 NLP test cases

In table 2, we detail the tests that we used for our NLP evaluation. Each test case contains a sentence that is processed by the NLP integration and the expected resulting data structure shown in the *Action*, *Objects* and *Spatial references* columns.

6.2 Placement strategies

Table 3 contains test cases for Broomrocket's placement strategies. Each test calls the placement strategy that should be evaluated with the specified number of cubes and the listed reference objects. The placement strategy is expected to return the stated placement volume and placement preference. Each volume is described with its bounding box as $(minX, maxX, minY, maxY, minZ, maxZ)$.

6.3 Extensibility

Broomrocket is designed as an algorithm that can be implemented in any 3D engine and programming language. The implementations for Blender 3.5 and Godot 4.2 serve as a showcase for our underlying algorithm. We structured the implementations in a way that makes them simple to adapt to other systems. Specifically, we provide two different implementation options to use the algorithm.

The first option is a pure Python implementation, which allows for further extension and improvements to the algorithm. The included *Engine* superclass is the integration point for any additional implementations. This class requires two functions to be provided: the *load_gltf* function for loading glTF [14] files into the current 3D scene, and the *list_objects* function for providing an abstract representation of all objects that are currently present in the scene. Further code may be necessary to provide a user interface integration, depending on the engine that is being implemented.

Subsequently, if a different NLP provider than SpaCy is required, the *NLPProvider* abstract class and its *parse* function, which converts a human sentence into structured data, should be implemented. For additional asset libraries apart from the default cube provider, a local asset folder, or Sketchfab, the *MeshProvider* abstract class and its *id*, *name*, and *find* methods need to be implemented. The mesh provider would also need to extend the *LoadableMesh* class and provide the *load_gltf* method.

In order to illustrate the effort required for further Python integrations, we provide the amount of code used for the Blender add-on in each category for the existing implementation in table 4. The source code contains region markers for each section for easier navigation.

The second option is to use our server which implements the NLP, object relation, and placement functionality engine-independently. The server sends the relevant instructions to the client and queries it for information

Test sentence	Action	Objects	Spatial references		
			Object	Location	Distance
Place 2 houses	place	house (2)			
Add a flower pot 5 meters behind the pools	add	flower pot	pool (plural)	behind	5 meters
Make 2 houses	make	house (2)			
Add a pool 5 meters behind the house	add	pool	house	behind	5 meters
Add two flower pots in front of the house	add	flower pot (2)	house	front	
There is a car and a ball to the left of the house in front of the tree	is	car, ball	house	left	
			tree	front	
There is a mountain in the background	is	mountain	background	in	
There is a billboard in the front	is	billboard	front	in	
There is a swing right of the house and the garden	is	swing	house, garden	right	
Place a lawnmower next to the house	place	lawnmower	house	next to	
Place a bench on the porch in front of the door	place	bench	porch	on	
			door	front	
Place a flower pot in front of the house	place	flower pot	house	front	
Place a house	place	house			
Make a house	make	house			
Behind the house and the forest there is a ball and a pool	is	ball	house	behind	
		pool	forest		
There is a ball and a pool behind the house and the forest	is	ball	house	behind	
		pool	forest		
Place a garden behind the house	place	garden	house	behind	
There is a pond in the back of the house	is	pond	house	back	
Add a cloud above the house	add	cloud	house	above	
Make another house	make	house			
Add rain below the cloud	add	rain	cloud	below	
Add rain under the cloud	add	rain	cloud	under	
Add a gnome between the houses	add	gnome	house (plural)	between	
Add a lamp on the table	add	lamp	table	on	
Add 50 trees and 2 balls around the house	add	tree (50), ball (2)	house	around	

Table 2. Broomrocket NLP integration tests

about the objects present in the 3D scene. Any programming language and 3D engine are suitable for a client-side implementation, as we demonstrate with our Godot plugin using GDScript.

7 CONCLUSION

Broomrocket is an object relation and placement algorithm for 3D scene creation with two concrete open source implementations using the popular 3D tools Blender or Godot as rendering engines. It combines 3D concepts with NLP and optional dedicated user asset libraries or an online 3D model distribution platform, where only objects that match the license chosen by the user are downloaded and properly attributed, for immediate usage

Algorithm	Test name	Objects	Reference objects	Placement volume	Preference
No	*single_mesh*	1		$(-0.5, 0.5, -0.5, 0.5, 0.0, 1.0)$	Y-orientation
	two_meshes	2		$(-0.5, 0.5, -1.0, 1.0, 0.0, 1.0)$	
Front	*single*	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 0.5)$	$(0.5, inf, -0.5, 0.5, -0.5, 0.5)$	Y-orientation Minimize X
	bottom	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 1.0)$	$(0.5, inf, -0.5, 0.5, -0.5, 1.0)$	
	double − refs	1	$(-0.5, 0.5, -1.0, 0.0, -0.5, 1.0)$ $(-0.5, 0.5, 0.0, 1.0, -0.5, 1.0)$	$(0.5, inf, -1.0, 1.0, -0.5, 1.0)$	
	double − targets	2	$(-0.5, 0.5, -0.5, 0.5, -0.5, 1.0)$	$(0.5, inf, -1.0, 1.0, -0.5, 1.0)$	
Behind	*single*	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 0.5)$	$(-inf, -0.5, -0.5, 0.5, -0.5, 0.5)$	Y-orientation Maximize X
	bottom	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 1.0)$	$(-inf, -0.5, -0.5, 0.5, -0.5, 1.0)$	
	double − refs	1	$(-0.5, 0.5, -1.0, 0.0, -0.5, 1.0)$ $(-0.5, 0.5, 0.0, 1.0, -0.5, 1.0)$	$(-inf, -0.5, -1.0, 1.0, -0.5, 1.0)$	
	double − targets	2	$(-0.5, 0.5, -0.5, 0.5, -0.5, 1.0)$	$(-inf, -0.5, -1.0, 1.0, -0.5, 1.0)$	
Left	*single*	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 0.5)$	$(-0.5, 0.5, -inf, -0.5, -0.5, 0.5)$	Y-orientation Maximize Y
	bottom	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 1.0)$	$(-0.5, 0.5, -inf, -0.5, -0.5, 1.0)$	
	double − refs	1	$(0.0, 1.0, -0.5, 0.5, -0.5, 0.5)$ $(-1.0, 0.0, -0.5, 0.5, -0.5, 0.5)$	$(-1.0, 1.0, -inf, -0.5, -0.5, 0.5)$	
	double − targets	2	$(-0.5, 0.5, -0.5, 0.5, -0.5, 1.0)$	$(-0.5, 0.5, -inf, -0.5, -0.5, 1.0)$	
Right	*single*	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 0.5)$	$(-0.5, 0.5, 0.5, inf, -0.5, 0.5)$	Y-orientation Minimize Y
	bottom	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 1.0)$	$(-0.5, 0.5, 0.5, inf, -0.5, 1.0)$	
	double − refs	1	$(0.0, 1.0, -0.5, 0.5, -0.5, 0.5)$ $(-1.0, 0.0, -0.5, 0.5, -0.5, 0.5)$	$(-1.0, 1.0, 0.5, inf, -0.5, 0.5)$	
	double − targets	2	$(-0.5, 0.5, -0.5, 0.5, -0.5, 1.0)$	$(-0.5, 0.5, 0.5, inf, -0.5, 1.0)$	
On	*single*	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 0.5)$	$(-0.5, 0.5, -0.5, 0.5, 0.5, inf)$	Y-orientation Minimize Z
	double − refs	1	$(-0.5, 0.5, -1.0, 0.0, -0.5, 0.5)$ $(-0.5, 0.5, 0.0, 1.0, -0.5, 0.5)$	$(-0.5, 0.5, -1.0, 1.0, 0.5, inf)$	
	double − targets	2	$(-1.0, 1.0, -1.0, 1.0, -0.5, 0.5)$	$(-1.0, 1.0, -1.0, 1.0, 0.5, inf)$	
Above	*single*	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 0.5)$	$(-0.5, 0.5, -0.5, 0.5, 0.5, inf)$	Y-orientation Minimize Z
	double − refs	1	$(-0.5, 0.5, -1.0, 0.0, -0.5, 0.5)$ $(-0.5, 0.5, 0.0, 1.0, -0.5, 0.5)$	$(-0.5, 0.5, -1.0, 1.0, 0.5, inf)$	
	double − targets	2	$(-1.0, 1.0, -1.0, 1.0, -0.5, 0.5)$	$(-1.0, 1.0, -1.0, 1.0, 0.5, inf)$	
Under	*single*	1	$(-0.5, 0.5, -0.5, 0.5, -0.5, 0.5)$	$(-0.5, 0.5, -0.5, 0.5, -inf, -0.5)$	Y-orientation Maximize Z
	double − refs	1	$(-0.5, 0.5, -1.0, 0.0, -0.5, 0.5)$ $(-0.5, 0.5, 0.0, 1.0, -0.5, 0.5)$	$(-0.5, 0.5, -1.0, 1.0, -inf, -0.5)$	
	double − targets	2	$(-1.0, 1.0, -1.0, 1.0, -0.5, 0.5)$	$(-1.0, 1.0, -1.0, 1.0, -inf, -0.5)$	

Table 3. Broomrocket placement algorithm tests

Category	LoC	Classes	Standalone functions
Broomrocket core	975	27	0
Placement strategies (incl. tests)	785	18	0
SpaCy NLP provider (incl. tests)	537	2	0
Dummy mesh provider	24	1	0
Local mesh provider	119	2	0
Sketchfab mesh provider	133	3	0
Blender engine integration	240	4	0
Blender UI integration	468	19	4
Total (incl. imports)	3,331	76	4

Table 4. Broomrocket for Blender components broken down into lines of code, classes, and functions

in game development and other 3D content creation. This work allows less technical users to start generating 3D scenes and includes commercial usage. By adding NLP to the toolchain, we have eliminated the additional learning curve of yet another syntax or yet another file format. Using standard text file imports over manual text

input, everyone can rapidly iterate on and reproduce their scene creation by editing narrative text. Broomrocket can speed up content creation time by not having to manually import and re-adjust objects, which is especially valuable for quick prototyping sessions and proofs of concept. It works best in combination with dedicated asset libraries as those can include higher quality meshes and metadata for the placement algorithm.

The general concept of Broomrocket can be applied to other 3D software including game engines, and due to the general extensibility of the concept and algorithms, new features can be added quickly and iteratively. It is a start into quick and efficient prototype design or release-ready environment design, depending on the datasets and inputs used. Other iterations of the concepts and ideas used in Broomrocket could focus on either integrations with different game engines, integrations with different 3D model distribution platforms, more fine-grained placement, atmospheric conditions such as post-processing filters as well as procedural materials and texturing, potentially combined with generative models.

Most importantly, more semantic combinations between material properties and words as well as attachment properties and semantic grouping are required to unleash the true potential of text-to-3D scene creation. In our work, we are not accounting for collisions, which works well for prototype level design and quick game design checks, but collisions would indeed be required for other applications such as character design, where meshes will inadvertently collide with each other, especially around the facial area. For real commercial applicability, more commands for scaling, rotating, editing, sculpting, painting, and more would need to be added.

REFERENCES

[1] George Anastassakis and Themis Panayiotopoulos. 2012. AN ARCHITECTURAL PATTERN FOR X3D-BASED VIRTUAL ENVIRONMENTS - An Object-oriented Approach. In *Proceedings of the International Conference on Computer Graphics Theory and Applications and International Conference on Information Visualization Theory and Applications - GRAPP, (VISIGRAPP 2012)*. INSTICC, SciTePress, 466–471. https://doi.org/10.5220/0003863404660471

[2] Olavo Da Rosa Belloc, Rodrigo B. D. Ferraz, Marcio Calixto Cabral, Roseli De Deus Lopes, and Marcelo Knörich Zuffo. 2012. Virtual Reality procedure training simulators in X3D. (2012), 153–160. https://doi.org/10.1145/2338714.2338741

[3] Steven Bird. 2006. NLTK: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive Presentation Sessions* (Sydney, Australia) *(COLING-ACL '06)*. Association for Computational Linguistics, USA, 69–72. https://doi.org/10.3115/1225403.1225421

[4] Blender Foundation 2023. *Blender website*. Retrieved June 30, 2023 from https://blender.org

[5] Epic Games 2023. *Unreal Engine 5: Procedural Content Generation Overview*. Retrieved November 11, 2023 from https://docs.unrealengine.com/5.2/en-US/procedural-content-generation-overview/

[6] ExplosionAI 2023. *SpaCy website*. Retrieved June 30, 2023 from https://spacy.io/

[7] gd3kr. 2023. *Blender GPT*. Retrieved November 11, 2023 from https://github.com/gd3kr/BlenderGPT

[8] Godot Engine 2023. *Godot website*. Retrieved November 1, 2023 from https://godotengine.org/

[9] GraphN 2023. *Dash*. Retrieved November 11, 2023 from https://www.polygonflow.io/

[10] Young-Seok Hwang, Jung-Sup Um, Biswajeet Pradhan, Tanupriya Choudhury, and Stephan Schlueter. 2023. How does ChatGPT evaluate the value of spatial information in the 4th industrial revolution? *Spatial Information Research* (2023). https://doi.org/10.1007/s41324-023-00567-5

[11] Cristina Improta. 2023. Poisoning Programs by Un-Repairing Code: Security Concerns of AI-generated Code. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE Computer Society, Los Alamitos, CA, USA, 128–131. https://doi.org/10.1109/ISSREW60843.2023.00060

[12] Heewoo Jun and Alex Nichol. 2023. Shap-E: Generating Conditional 3D Implicit Functions. arXiv:2305.02463 [cs.CV]

[13] Fahad Khan, Kashif Irfan, Razzaq Saad, Maqbool Fahad, Farid Ahmad, and Rao Anwer. 2008. Using VRML to Build a Virtual Reality Campus Environment. *Lecture Notes in Engineering and Computer Science* 2170 (07 2008).

[14] Khronos Group 2023. *glTF Overview - The Khronos Group Inc*. Retrieved November 1, 2023 from https://www.khronos.org/gltf/

[15] Zhang Lin, Chen Zhixing, and Zhao Chunxiao. 2010. Research and modeling the ancient architecture system in VRML. In *2010 International Conference on Computer and Communication Technologies in Agriculture Engineering*, Vol. 3. 587–590. https://doi.org/10.1109/CCTAE.2010.5544864

[16] Hartmut Luttermann and Manfred Grauer. 1998. VRML-basierte Präsentation raum-zeitlicher Geschäfts- und Wissenschaftsdaten mit WWW-Browsern. In *Informatik '98*. Springer Berlin Heidelberg, Berlin, Heidelberg, 57–66.

[17] Alexey Mazalov, Bruno Martins, and David Matos. 2015. Spatial Role Labeling with Convolutional Neural Networks. In *Proceedings of the 9th Workshop on Geographic Information Retrieval* (Paris, France) *(GIR '15)*. Association for Computing Machinery, New York, NY,

USA, Article 12, 7 pages. https://doi.org/10.1145/2837689.2837706
[18] Alex Nichol, Heewoo Jun, Prafulla Dhariwal, Pamela Mishkin, and Mark Chen. 2022. Point-E: A System for Generating 3D Point Clouds from Complex Prompts. arXiv:2212.08751 [cs.CV]
[19] Ben Poole, Ajay Jain, Jonathan T. Barron, and Ben Mildenhall. 2022. DreamFusion: Text-to-3D using 2D Diffusion. arXiv:2209.14988 [cs.CV]
[20] Procedural Worlds 2023. *GeNa Pro*. Retrieved November 11, 2023 from https://www.procedural-worlds.com/products/professional/gena-pro/
[21] Sketchfab 2023. *Sketchfab website*. Retrieved June 30, 2023 from https://sketchfab.com/
[22] Something Pink 2024. *Broomrocket: Open Source Text to 3D Scene Creation*. Retrieved February 9, 2024 from https://something.pink/software/broomrocket/
[23] VRML Specification 2023. *VRML97*. Retrieved June 30, 2023 from https://www.web3d.org/documents/specifications/14772/V2.0/
[24] Zhengyi Wang, Cheng Lu, Yikai Wang, Fan Bao, Chongxuan Li, Hang Su, and Jun Zhu. 2023. ProlificDreamer: High-Fidelity and Diverse Text-to-3D Generation with Variational Score Distillation. arXiv:2305.16213 [cs.LG]
[25] World Machine Software 2023. *World Machine*. Retrieved November 11, 2023 from https://www.world-machine.com/
[26] Hongwei Yi, Chun-Hao P. Huang, Shashank Tripathi, Lea Hering, Justus Thies, and Michael J. Black. 2022. MIME: Human-Aware 3D Scene Generation. arXiv:2212.04360 [cs.CV]
[27] Livia Ştefan, Sorin Hermon, and Marina Faka. 2018. Prototyping 3D Virtual Learning Environments with X3D-based Content and Visualization Tools. *BRAIN. Broad Research in Artificial Intelligence and Neuroscience* 9 (05 2018).