applied sciences

MDPI

*Article*

# New Game Artificial Intelligence Tools for Virtual Mine on Unreal Engine

**Fares Abu-Abed [1] and Sergey Zhironkin [2,3,4,*]**

1 Department of Electronic Computers, Faculty of Information Technologies, Tver State Technical University, 22 Afanasiya Nikitina Emb., 170026 Tver, Russia; aafares@mail.ru
2 Department of Open Pit Mining, T.F. Gorbachev Kuzbass State Technical University, 28 Vesennya St., 650000 Kemerovo, Russia
3 Department of Trade and Marketing, Siberian Federal University, 79 Svobodny Av., 660041 Krasnoyarsk, Russia
4 School of Engineering Entrepreneurship, National Research Tomsk Polytechnic University, 30 Lenina St., 634050 Tomsk, Russia
* Correspondence: zhironkinsa@kuzstu.ru

**Abstract:** Currently, the gamification of virtual reality for training miners, especially for emergencies, and designing the extraction of minerals in difficult technological conditions has been embodied in the Virtual Mine software and hardware. From a software development point of view, Virtual Mine is indistinguishable from other virtual reality games, and this offers a chance to use the potential of rapidly developing game software in mining, including engines, 3D modeling tools, audio editors, etc., to solve a wide range of game development tasks. The chosen direction will optimize the work of developers by providing a tool for developing game artificial intelligence to solve problems that require implementing the behavior of game agents without using a rigidly defined choice of scenarios or chains of these scenarios. The aim of the work is to expand the possibilities of working with game artificial intelligence on the Unreal Engine game engine to make it more functional. As a result, a tool has been obtained that can be used to optimize the time and improve the quality of the development of game artificial intelligence for Virtual Mine using flexible development approaches. The asset editor was developed, application modes and their working tabs were defined, and a graphical node system for the behavioral graph editor was created. A system for executing a behavioral graph is given; algorithms for its operation and features for executing nodes of a behavioral graph are presented.

**Keywords:** virtual mine; IT industry; game engines; 3D modeling tools; artificial intelligence; behavior of game agents; unreal engine; virtual reality

check for updates

**Citation:** Abu-Abed, F.; Zhironkin, S. New Game Artificial Intelligence Tools for Virtual Mine on Unreal Engine. *Appl. Sci.* **2023**, *13*, 6339. https://doi.org/10.3390/app13106339

Academic Editor: Paolo Renna

Received: 29 March 2023
Revised: 17 May 2023
Accepted: 20 May 2023
Published: 22 May 2023

## 1. Introduction

The concept of gamification of virtual reality tools for miners training, ensuring the safety of workers, and designing mining operations was launched in the early 2000s. Initially, gamification was understood as a direction for the development of simulation systems for training personnel, assessing their competencies, developing knowledge management, and implementing best practices [1].

The first attempts to build virtual 3D models of mines with an interface similar to computer games were not aimed at creating a fully interactive digital reality but rather animation software controlled by mouse and keyboard [2].

Subsequently, since the beginning of the 2010s, virtual reality tools such as VR glasses and hand-held manipulators, volumetric motion sensors, and powerful computing systems have started to be used [3]. Here an and below the terminology used in this article is explained in Appendix A.

To date, virtual reality game models in the mining sector incorporate the advanced achievements of Industry 4.0 (artificial intelligence, digital 3D models, and smart sensors) and are part of the Mining 4.0 technology platform [2].

Mining engineers training using virtual reality is gaining importance [4]. This process not only improves the quality of training of mining experts but also significantly reduces the number of possible emergency and pre-emergency situations, taking into account the experience gained and practical training for the application of proactive measures in underground space as close as possible to real scenarios [2–4].

In particular, virtual gaming systems are widely used to conduct introductory briefings and tours of underground workings, prepare firefighting activities, and eliminate the consequences of accidents underground, which makes it possible to reduce occupational injuries by 1.5–2.5 times [5]. They also improve the design and management of disturbed land reclamation through high-level personnel training [6].

The use of gaming virtual simulator systems in mines is associated with the difficulty of recreating the real situation since geotechnics and technologies are developing quite rapidly. Nevertheless, virtual gaming technologies allow, at minimal cost, the automatic launch of risk management and competency adjustment procedures [7].

The analysis of literary sources showed the high contribution of scientists and practitioners in the development of computer applications, the use of virtual and augmented reality technologies, as well as artificial intelligence objects [8–18]. These works have covered many diverse scientific and technical fields, from the development of object detection systems [19], education [20,21], cinematography [22], equipping scientific laboratories [23], solving tourism problems [24], modeling and optimization [25], to gaming applications [26–28]. However, not one work presented the process of expanding the possibilities of developing game artificial intelligence. This is precisely the main goal disclosed in the presented article.

Artificial intelligence in video games is a system that determines the believable behavior of in-game characters controlled by a computer. The presence of a believable game AI is one of the necessary conditions for the player's quality immersion in the virtual world. Game AI is rarely associated with any kind of deep intelligence but rather with the illusion of intelligence. Game AI developers often try to create believable human behavior, but the actual intelligence that can be programmed is quite limited and painfully fragile. Expectation plays a huge role in how people perceive the world, and if expectations are not properly managed, then even truly intelligent behavior on a human level can be perceived as incompetent and clearly inhuman. Game AI should focus on one thing and one thing only: enabling developers to create an engaging experience for the player. Every technique used, every trick played, and every algorithm coded must support this single goal.

Each game is unique, and the needs of AI in games vary greatly. At the same time, the goals of game AI tend to have much more in common with the cinematic view of artificial life than with the classical academic view of AI. Like cartoons, games are created for entertainment. Like cartoons, games are not about maximizing success, cognitive modeling, or real intelligence but rather about telling a story, creating an experience, and creating the illusion of intelligence. In some cases, the methods we need to create this illusion can be taken from academic AI, but in many cases they are different. We use the term "game AI" to describe AI that focuses on creating the appearance of intelligence and creating a specific experience for the viewer rather than creating the true intelligence that exists in humans. Game AI is not real AI but just a simulation of having one.

In most game projects, when writing game AI systems, they use approaches that have not changed over a long stage in the development of the game industry since the appearance of the first virtual opponents in 3D games [29,30]. These approaches make it possible to conveniently compare the behavior required from AI with a strictly defined set of conditions that cause this behavior, but the behavior of game AI implemented using these approaches is too static.

The static behavior of game AI no longer satisfies the requirements of modern players, and the implementation of game AI capable of providing for many game situations is difficult to design and expensive to develop, as a result of which developers have to find a balance between providing for all possible game situations and the cost of developing game AI.

Therefore, there is a growing need in the gaming industry for AI that is simple to design yet able to handle complex behavior—that is, an AI capable of unpredictable actions and making decisions in situations not necessarily envisaged by the AI designer.

To solve this problem, there is a new, not yet widely adopted, paradigm for implementing a key component of game AI, namely the game AI decision system—the utility-based decision system. The utility-based system paradigm proposes to implement decision-making systems that evaluate utility, or, in other words, the priority of choice, rather than being guided by rigidly prescribed conditional structures when making decisions for further behavior, as is customary in traditional approaches. An in-depth analysis of literary sources showed the lack of a tool for solving problems that require the implementation of the behavior of game agents without using a rigidly defined choice of scenarios or chains of these scenarios.

When developing AI systems based on utilities, there is a need for an approach that, in principle, can implement the required behavior with a decrease in labor intensity for the software developer. This will make it possible to create a highly competitive tool for the mass introduction of game teaching and design methods in virtual mines. The tool presented in the article is necessary for solving a certain class of problems, namely the use of an approach based on unities to develop the behavior of game AI in Unreal Engine 5 in order to make game methods for teaching and training miners as realistic as possible.

## 2. Materials and Methods

### *2.1. Gamification Technologies in Mining Simulation by Virtual Reality*

In mining, various interactive systems for virtual reality are being developed, including those based on game engines. Simpler systems allow visualizing virtual reality, conceptually reflecting the essence of processes in mine workings and human participation in managing them.

For example, virtual 3D systems are being used to train workers on EXP360's ProExpVR platform, using realistic datasets that strike a balance between network performance and VR video optimization [31].

The simulated zones exactly repeat the technological schemes used for organizing tunneling and excavation works (Figure 1) [32].

The use of game simulation in virtual 3D modeling of emergency situations has positively proven itself in training personnel for acting in emergency situations when virtual robots (bots) controlled by artificial intelligence play the role of partners who violate safety regulations or rescuers, as well as breaking equipment (Figures 1 and 2) [32,33].

The use of the Virtual Mine VR Professionals by DT Consulting for collective firefighting game training is shown in Figure 3 [33].

There is a practice of using widely used gamification tools, such as Microsoft Kinect for Xbox, in mining, which allows controlling a digital avatar in a virtual environment using spoken commands, body gestures, and facial mimics [34]. This includes building a map of the working's depth and separating a person from the background, identifying individuals, building a virtual human skeleton, identifying simple gestures and tracking movements, and providing feedback using an accelerometer. The gamification of a person moving through a virtual mine using Microsoft Xbox and Kinect is built on open (OpenNI, Open CV) or closed solutions (Kinect SDK); to implement physical effects, the PhysX game engine is used [34,35].

The following block diagram is proposed for the simulation of the game training system of a virtual mine (Figure 4).

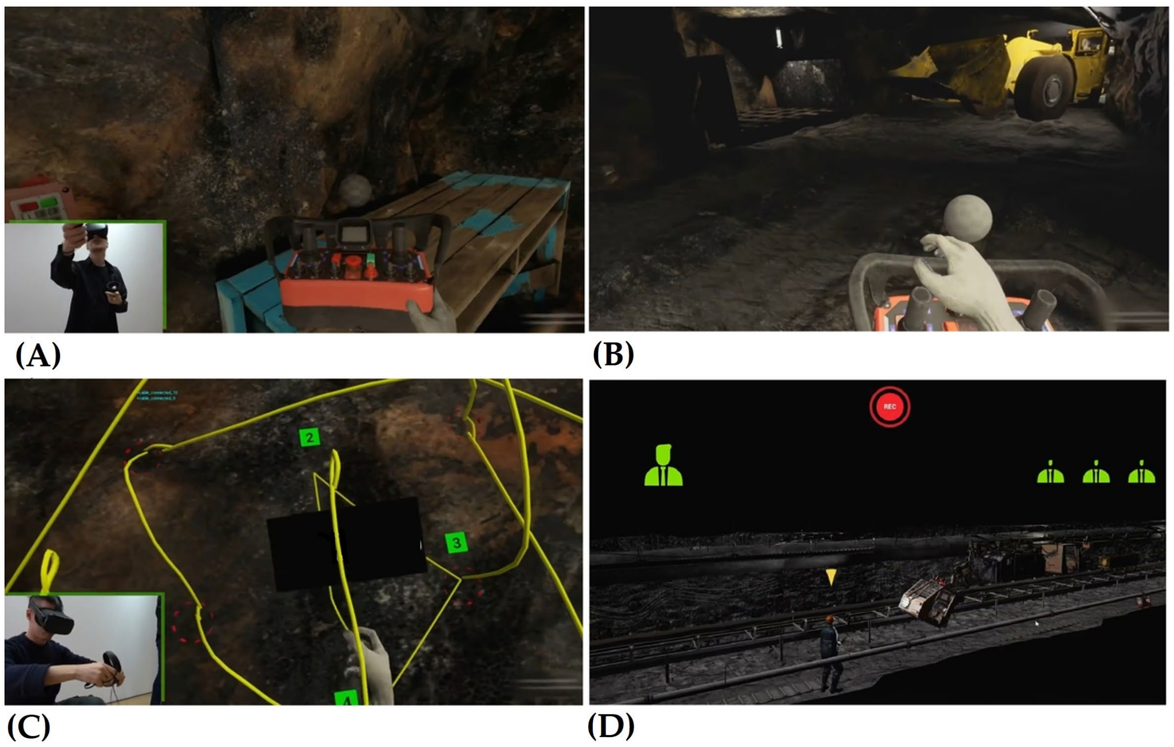


**Figure 1.** Simulation of actions in Virtual Mine on the ProExpVR platform: (**A**) preparation for blasting; (**B**) remote control of the mine loader; (**C**) installation of an explosion initiation network; (**D**) mine transport accident model (Reprinted from Ref. [32]).

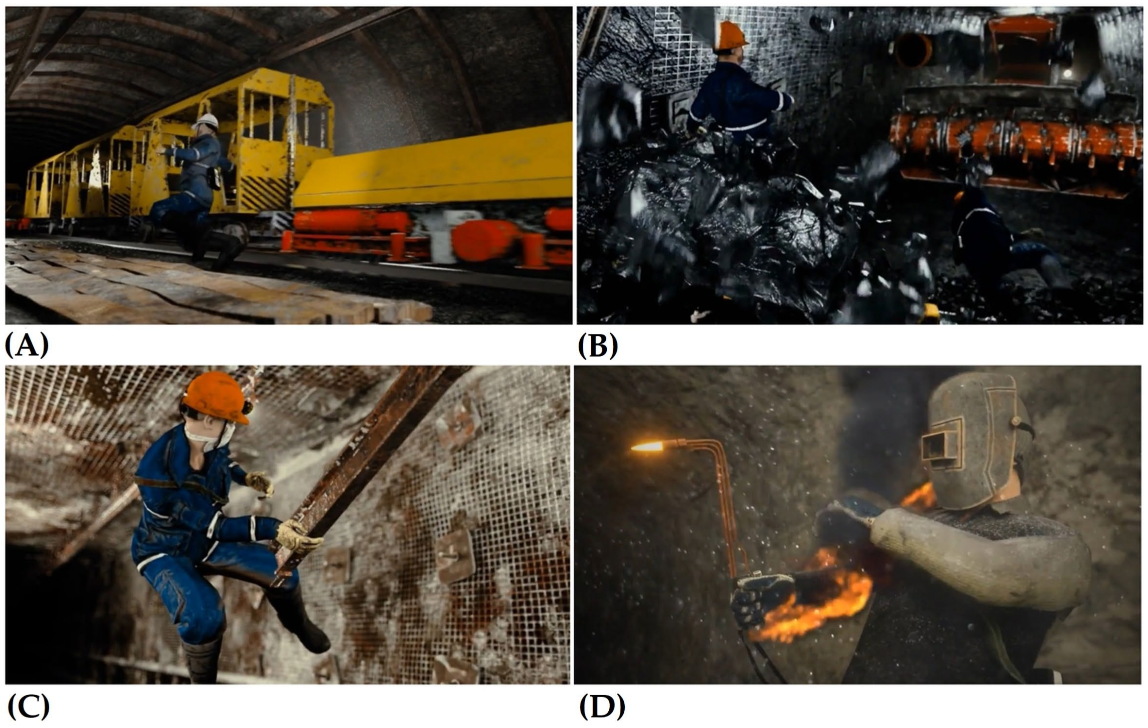


**Figure 2.** Game virtual 3D models of coal and ore mine accidents (ProExpVR platform): (**A**) trolley hooking and dragging; (**B**) roof collapse; (**C**) fall from a height; (**D**) ignition during welding (Reprinted from Refs. [32,33]).

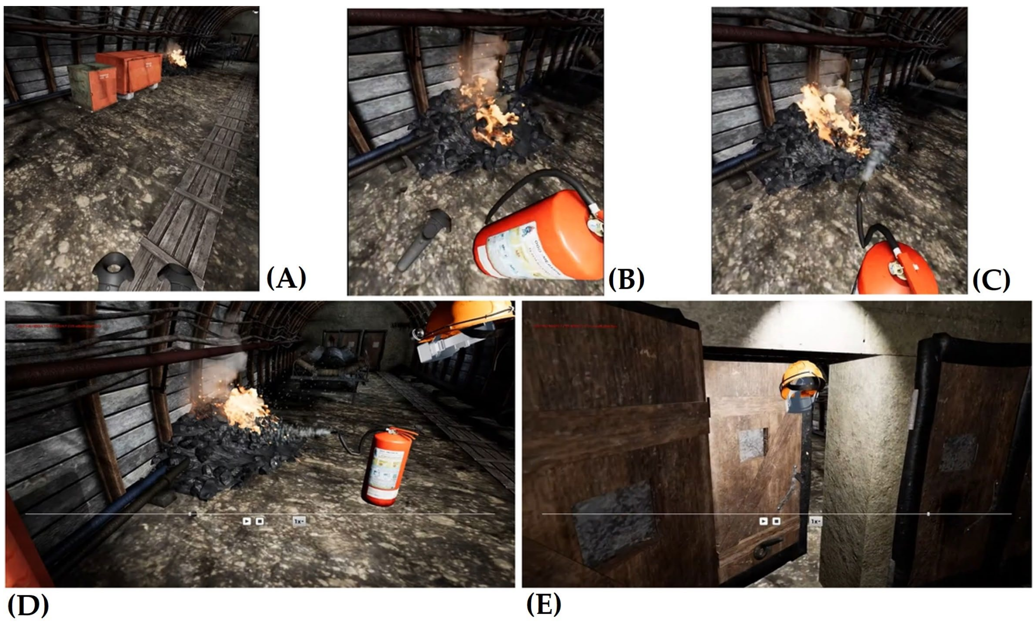


**Figure 3.** Coal firefighting training in an underground mine using Virtual Mine VR Professionals: (**A**) approaching the source of fire; (**B**) preparing the fire extinguisher; (**C**,**D**) fire extinguishing from different angles; (**E**) leaving work (Reprinted from Ref. [33]).

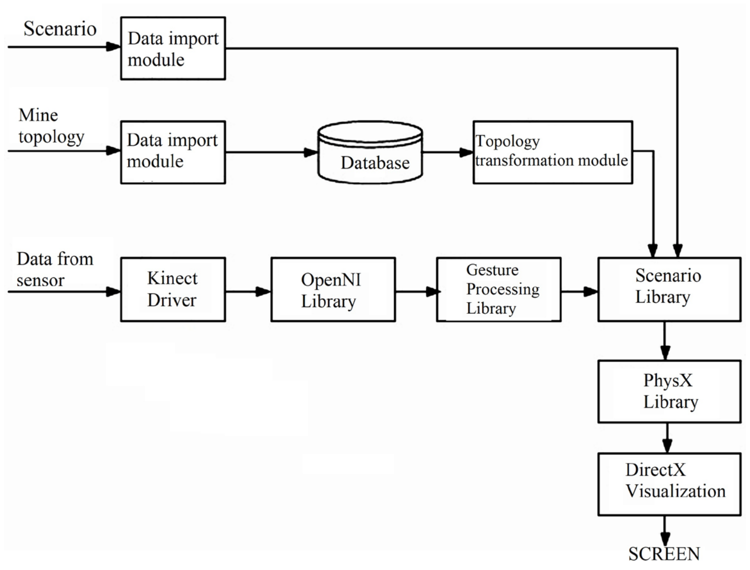


**Figure 4.** The structure of the program modules of the Virtual Mine system on the Kinect gaming platform (Reprinted from Ref. [35]).

To develop the topology of the mine (geometry of workings and their spatial orientation) and scenarios (artifacts of the virtual world), three-dimensional models in AutoCAD format processed by Blender software are used. The resulting topology is stored in the FireBird database; to implement a variety of physical effects, the PhysX game engine is used (person positioning data are read through the driver and OpenNI library)—Figure 5.

Another example of a virtual reality complex for the gamification of training future engineers, including miners, is the use of the Godot engine at the University of Žilina, which has proven itself for industrial applications [P]. The HTC Vive Pro headset was chosen as a display and control device for the virtual reality game (Figure 6 [36]).

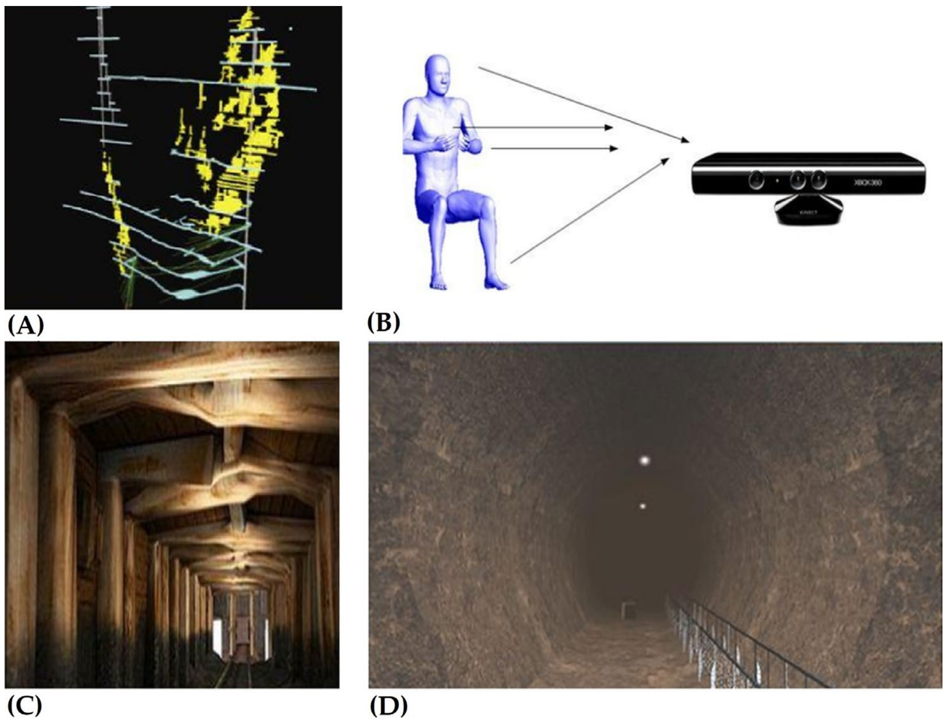


**Figure 5.** Construction and use of a virtual mine model using Kinect for Microsoft Xbox: (**A**) a three-dimensional model of a mine working in AutoCAD format; (**B**) transferring digital avatar control gestures using Kinect; (**C**) game visualization of the mine working model after Blender processing; (**D**) an example of driving through a mine working based on the PhysX game engine (Reprinted from Refs. [34,35]).

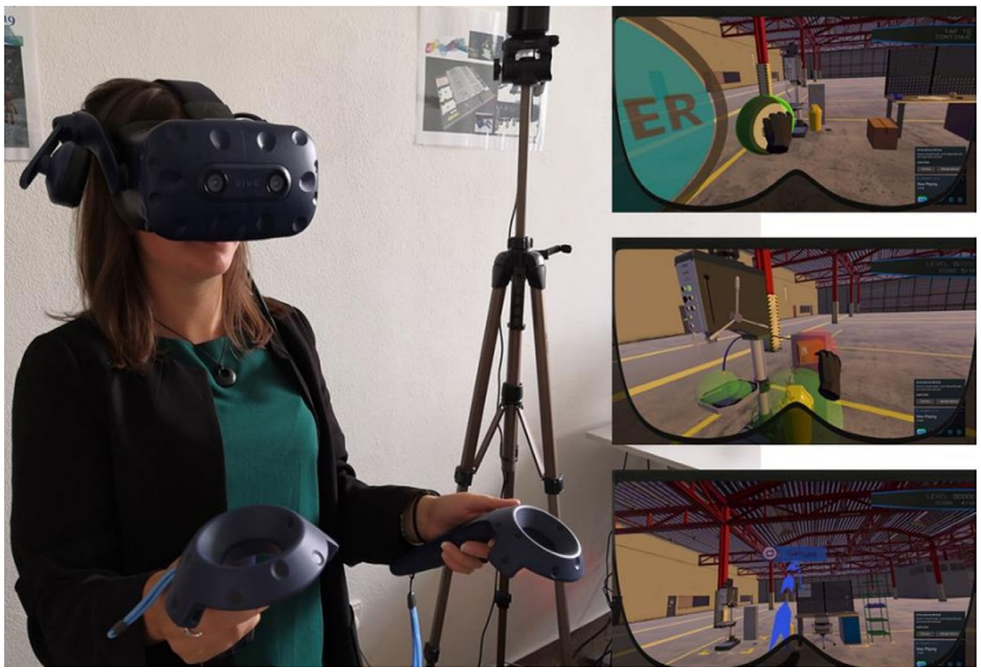


**Figure 6.** Application of gaming virtual reality based on Godot engine for teaching students at the University of Žilina (Reprinted from Ref. [36]).

The range of research on the use of artificial intelligence for virtual assistants and multimedia databases should also be noted [37–41].

The variety of virtual reality gaming simulation tools available for training mine personnel, ensuring labor safety [42–45], and designing production processes makes it possible to explore the best software systems and the possibilities of their application.

### *2.2. Goal of the Work*

Expanding the capabilities of working with game artificial intelligence on the Unreal Engine game engine.

Utility-based artificial intelligence uses a heuristic function to calculate the utility of each option, and then that utility is used to make decisions. This is usually performed by selecting the option with the highest utility. The advantage of utility-based AI is that it allows the AI to consider the subtle nuances of the situation when making a decision.

Virtual 3D systems are being used to train workers on EXP360's ProEx-118 pVR platform. However, such systems are, in fact, ready-made solutions, while the tool developed by the authors is addressed to software developers using virtual reality technologies and not to end users. In fact, the authors' approach allows editing Game Artificial Intelligence Tools on Unreal Engine 5 as mining technologies develop and the class of tasks to be solved expands.

In Unreal Engine, a module is a single unit of C++ code with an accompanying C# assembly file. The module corresponds to a dynamic library.

There are many different menu bars and toolbars in Unreal Engine, which can be composed of individual Slate elements or generated using FMenuBuilder, and which can be composed of individual interface commands, command lists, or individual interface actions.

The base class for developing an asset editor is FAssetEditorToolkit, which contains the basic functions for developing an asset editor. FWorkflowCentricApplication is a class derived from FAssetEditorToolkit that allows you to create an asset editor that allows multiple application modes.

The Blueprint debugger provides control over game execution during PIE and SIE sessions. The controls become available on the toolbar when the game is running. Different debug controls appear depending on the type of Blueprint being debugged and the current state of the debug session. Some controls only become active when needed, such as when a breakpoint is hit.

Widget Reflector is a tool that allows developers to define and debug Slate widgets. The Unreal Editor interface is built using the Slate UI framework. Widget Reflector allows developers to define the Slate API used to display various widgets in a toolbox.

In Unreal MarketPlace, there is no plug-in for the game's artificial intelligence development tool based on utilities for virtual mines. Because of this, the process of preparing miners loses competition in terms of virtual reality gaming tools. Therefore, the solution proposed by the authors allows creating a large number of different game models of virtual mines for the first time in relation to various types of minerals, mining, and geological conditions.

## **3. Results**

### *3.1. Combining Code and Content in Unreal Engine*

#### 3.1.1. UE/C++ Module

In Unreal Engine, a module is a single unit of C++ code with an accompanying C# assembly file. A game project can consist of one or more modules, just like a plugin. The module corresponds to a dynamic library, although by default, in ready-made assemblies, all code is linked into one executable file. Using modules is a good code organization practice to reduce project build time and set up loading and unloading processes for systems and code.

Any module must have three files. The first is the <Module name>.h header file for this module, usually including the Engine.h file. In addition, this file contains the interface for the module. Where you can register anything. The second required file is the <Module name>.cpp file. At the end of this file, there must be a macro IMPLE-

MENT_GAME_MODULE. The third required file is <Module name>.Build.cs. This file connects the modules necessary for the developed module to work. After creating these files, you need to put them in a folder with the name of the module in the Source folder.

In order for a class or structure from a connected module to be used. Need to:

1. Connect the header file of this class or structure.
2. When defining classes and structures that you want to use from other modules, you need to write <module name in capital letters>_API.

#### 3.1.2. Unreal Engine Plugin

In Unreal Engine, plugins are collections of code and data that developers can easily enable or disable in the editor on a per-project basis. Plugins can add gameplay functionality at runtime, change built-in Engine functions (or add new ones), create new file types, and extend the editor's capabilities with new menus, toolbar commands, and submodes. Many existing UE4 subsystems have been designed to be extensible via plugins.

Plugins with code will have a Source folder. This folder will contain one or more module source directories for the plugin. Note that although plugins often contain code, this is not actually a requirement.

A plugin should be thought of as a project that can be inserted into another project. A plugin, like a project, consists of modules and has its own project file, which has the uplugin extension. To connect a plugin to a project, it must be placed in the <project directory>/plugins folder.

Plugins with code will have a Binaries folder containing the compiled code for that plugin, and temporary build product files will be stored in a separate Intermediate folder in the plugin directory.

Plugins can have their own Content folder containing asset files specific to that plugin. Plugin configuration files must be placed using the same convention as other configuration files:

- Engine plugins:[PluginName]/Config/Base[PluginName].ini
- Game plugins:[PluginName]/Config/Default[PluginName].ini

Unreal Engine finds your plugin by looking for uplugin files on disk. These files are called plugin descriptors. These are text files containing basic information about the plugin. Plugin descriptors are automatically detected and loaded by the engine, editor, and UnrealBuildTool (UBT) each time these programs are launched.

Figure 7 shows the structure of the Unreal Engine modules:

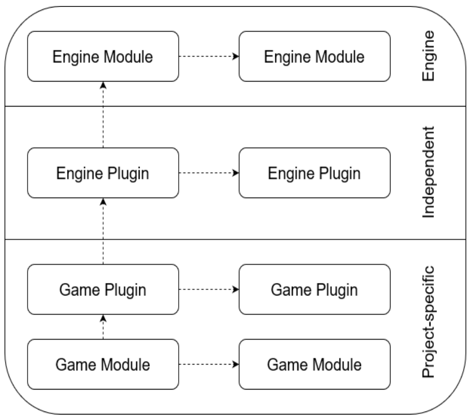


**Figure 7.** Unreal Engine Module Structure.

### *3.2. Unreal Engine Command System*

There are many different menu bars and toolbars in Unreal Engine, which can be composed of individual Slate elements or generated using FMenuBuilder, and which can be composed of individual interface commands, command lists, or individual interface actions. Each interface command is an instance of FUICommandInfo that can be launched in various ways, for example, using keyboard shortcuts or buttons as part of a menu or toolbar (Figure 8).

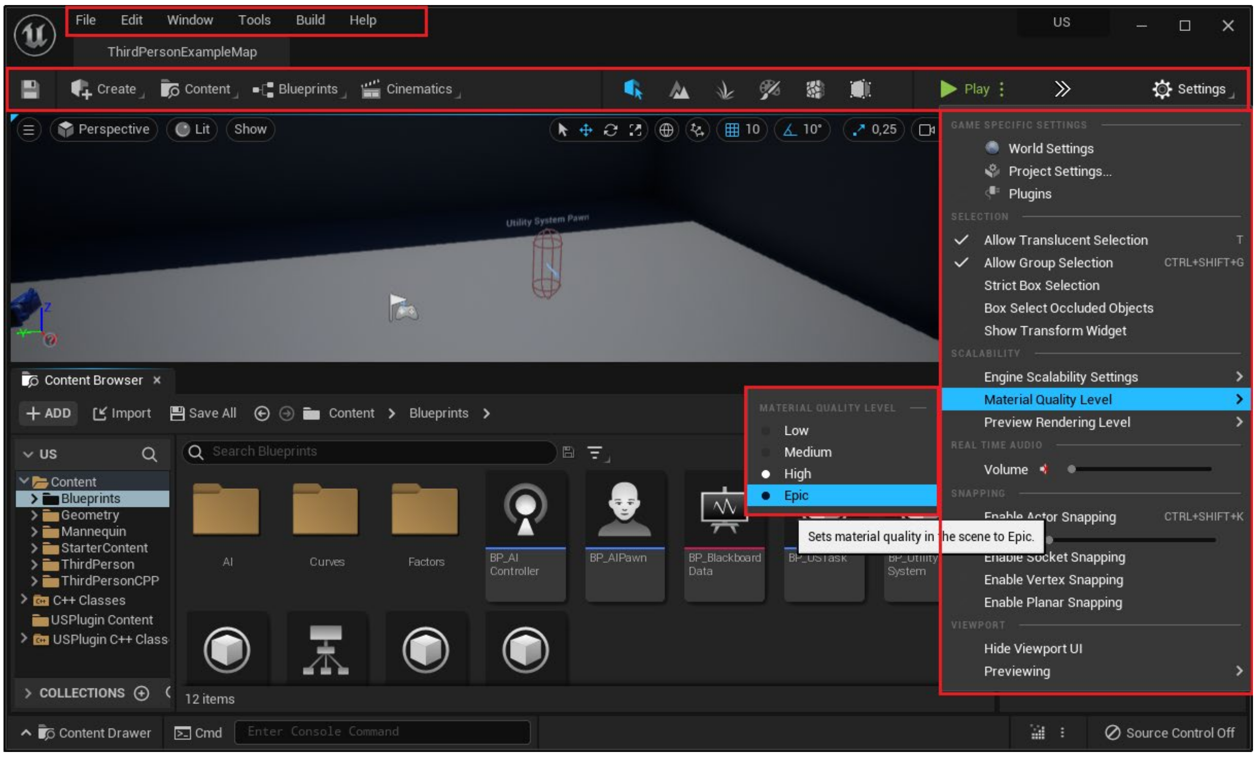


**Figure 8.** Various Unreal Engine Menu Bars and Toolbars.

### 3.2.1. Commands and Command Groups

FUICommandInfo is a class that provides basic information about a command. An instance of FUICommandInfo is used to bind menu execution events. Menu execution events are bound by passing a delegate with an associated execution function.

TCommands<T> is a class that functions as a collection of commands for declaring various FUICommandInfos and registering FUICommandInfos. For command registration to occur, you must override the RegisterCommands virtual method. To register each command, you must use the UI_COMMAND macro. The UI_COMMAND macro associates the declared command with its description, the type of button that will be displayed in the menu (regular button, radio button, etc.), as well as the keys or key combination that can be used to execute the command.

FUICommandList is a list of commands; each instance of FUICommandInfo must be added to the list via the MapAction method on FUICommandList. The key difference between a command list (FUICommandList) and a command collection (TCommands<T>) is that a command list is a combined sequence of commands that can be added to the entire menu in a certain order, and a command collection is a set of independent commands, each of which is added to the menu separately.

### 3.2.2. Expanding the Unreal Engine System Menus

There are three main menu building elements in Unreal Engine:

- FMenuBarBuilder—Primarily used to extend the main menu;
- FMenuBuilder—Mainly used for Mainly used for menu extension;
- FToolBarBuilder—Extends the toolbar.

There are other menu builders, but the ones listed above are the highest in the inheritance hierarchy of menu builders.

The inheritance hierarchy of menu builders is shown in Figure 9.

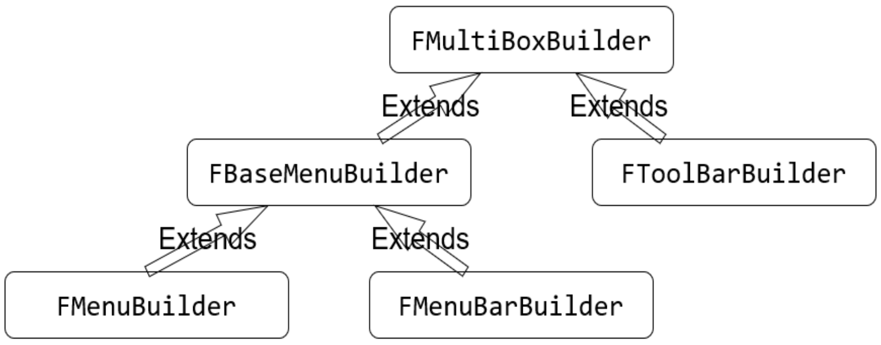


**Figure 9.** Menu Builder Inheritance Hierarchy.

The main differences between the builders listed above are the presence or absence of divisions and submenu sections, as well as the appearance of the Slate widgets generated by the builder.

The FMenuBarBuilder only allows the creation of a menu bar without any divisions or display groups of widgets.

FMenuBuilder allows you to create complex menus divided into different submenu sections.

FToolBarBuilder allows you to create a menu line with menu items divided into blocks of items.

Generalized Menu Expansion Process

An instance of UICommand Info should be placed as a property of the TCommands<T> template class and created there; then one FUICommandList will be mapped to one FUICommandInfo with one FUIAction (what the command does). Finally, somewhere in the editor, an "extension" will be added to the interface, specifying the "extension," FUICommandList, and what FUICommandInfo interface elements will be added.

It can be represented as the following figure (Figure 10):

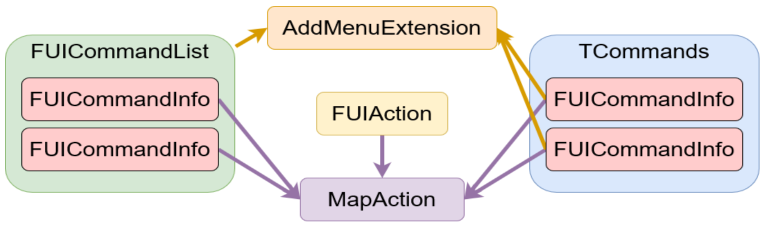


**Figure 10.** Menu expansion process.

As Unreal Engine editing expands, the interface created by the interface can be added to menus and toolbars. It is also possible to add menus to other window interfaces.

There are many different menu bars and toolbars in Unreal Engine. All of them are extensible with FExtend. FExtend provides AddMenuExtension for the Menu extension, AddMenuBarExtension for the MenuBar extension, and AddToolBarExtension for the ToolBar extension. Finally, FExtend is managed in the FLevelEditorModule.

### *3.3. UE4 Custom Asset Type Action*

#### 3.3.1. Defining an Asset Class

To define an asset class, you need to create a new class derived (inherited) from the base class for all objects used in Unreal Engine—UObject.

#### 3.3.2. Defining Custom Asset Type Actions

Asset Type Actions (IAssetTypeActions) is an interface that provides actions and other information about asset types.

The asset type base actions (FAssetTypeActions_Base) is a class that provides a default implementation of the methods of the IAssetTypeActions interface.

The type action overload allows defining the name, color, and category for the supported asset, as well as some system actions applicable to the asset, such as the OpenAssetEditor method, which is called when the asset is opened in the Unreal Engine Asset Browser window and opens the editor for the selected asset.

In order for the actions of the asset type to be applied, you must:

3. Specify an asset class as a supported class;
4. Register the asset type action class in the FAssetToolsModule.

#### 3.3.3. Asset Factory

In order for Unreal Engine to recognize the custom asset and create it, you need to add a class that inherits from UFactory. Then, just like the class that implements the IAssetTypeActions interface for the factory, you must specify the asset class as a supported one and define the FactoryCreateNew method, which will create the object of the supported asset. The factory class must be in the same module as the asset class. Unreal Engine automatically captures all UFactory declarations during editor loading, so there is no need to manually register factories.

### *3.4. UE Asset Editor Class Definition*

#### 3.4.1. FAssetEditorToolkit

The base class for developing an asset editor is FAssetEditorToolkit, which contains the basic functions for developing an asset editor. The structure is shown below in Figure 11.

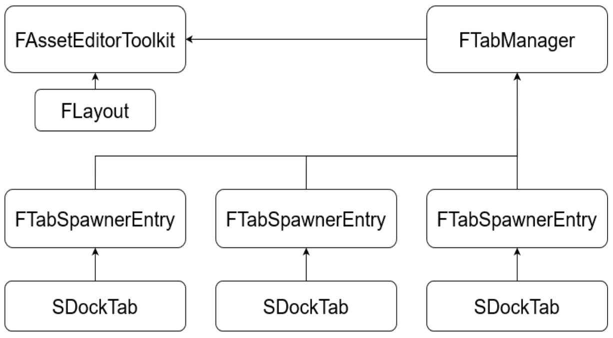


**Figure 11.** FAssetEditorToolkit block diagram.

The resource editor consists of several tabs. The basic process of creating a resource editor is:

- Derivate an editor class from FAssetEditorToolkit;
- Create a layout for your editor;
- Registering and creating tabs using FTabManager.

The key functions of the FAssetEditorToolkit subclass that must be called or overridden are as follows:

- InitAssetEditor—Initializes this asset editor. Called immediately after the editor is built;
- RegisterTabSpawners—Registers tab factories.

#### 3.4.2. FWorkflowCentricApplication

Some asset editors, such as FBehaviorTreeEditor or FPersonaAssetEditorToolkit, may have multiple application modes that allow switching between tab groups in different modes. To conveniently achieve this, Unreal Engine provides a class derived from the FAssetEditorToolkit called FWorkflowCentricApplication. Figure 12 shows the block diagram of the FworkflowCentricApplication.

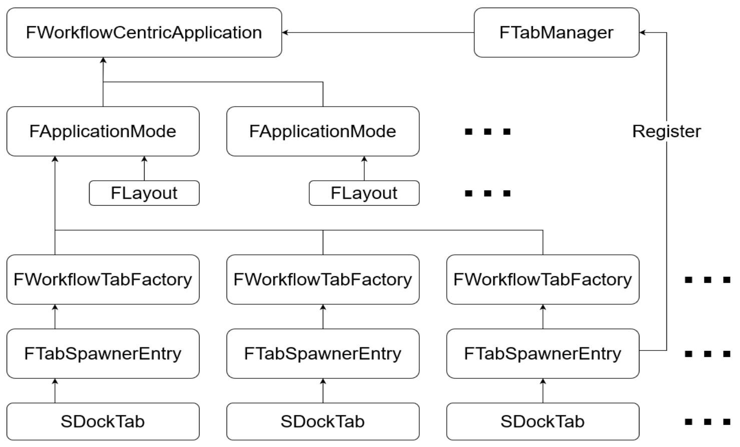


**Figure 12.** Block diagram of FWorkflowCentricApplication.

- An asset editor inherited from FWorkflowCentricApplication can have multiple application modes (FApplicationMode);
- Each FApplicationMode has its own layout and tabs;
- Each FApplicationMode has multiple FWorkflowTabFactory;
- Each FWorkflowTabFactory will create a tab for a different purpose.

## *3.5. Defining a Custom Asset Editor*

### 3.5.1. Application Mode

Application mode (FApplicationMode) is a class that groups the menu, toolbar, and working tabs of the application for this mode of the asset editor. When FWorkflowCentricApplication sets any application mode as current, it cleans up the menu and toolbar extensions completely and then installs the menu and toolbar extensions from application mode to the asset editor. The layout of working tabs is obtained, and a complete rebuild of the Slate component of the asset editor is performed.

### 3.5.2. Spawning the Working Tabs of the Asset Editor

To create workflow tabs, you need a class inherited from FWorkflowTabFactory. In the class constructor, you must set the identifier for the future table. The tab ID is a simple text field. The tab ID is used to further lay out the tabs in the tab manager, so it is good design practice to create a data class that stores the static tab IDs for each application mode.

For tabs, you can specify names, an icon, a textual description of the tab, and whether a tab of this type will be the only instance in the editor. The virtual method CreateTabBody is responsible for the appearance of the tabs and should return a general reference to the widget's Slate object.

### 3.5.3. Application Tab Layout

The layout of application mode work tabs is performed by defining a TabLayout in application mode. The tab manager (FTabManager) is used to create the tab layout. The layout generated by the tab manager implements the Builder design pattern, thanks to which the layout can be easily modified by extension methods by adding new areas and their separators. Each tab area can be thought of as a container of tabs, either horizontally or vertically. Tab containers can be placed inside other containers, and you can also set the relative scale of areas within the same container, which allows you to conveniently adjust the scaling and tab layout for each application mode.

### 3.5.4. Extending the Toolbar

To add or remove toolbar items, use the array of toolbar extenders (ToolbarExtenders) in the FAssetEditorToolkit. To extend the toolbar, follow these steps:

5. Form a set of commands (FUIAction) that will be executed. To do this, in a class derived from FAssetEditorToolkit, add new commands to ToolkitCommands and specify executing methods for them.
6. Generate a delegate for each extension (FToolBarExtensionDelegate) that accepts a hard link to the toolbar builder (FToolBarBuilder). The delegate uses the toolbar builder to add the appropriate keys to it.
7. Form an expander (Fextender) in the form of a common pointer.
8. Add an extension to the extender by specifying a set of commands and a delegate.

### *3.6. Unreal Engine Development Tools*

The developer tools provide you with several information-gathering tools, such as debuggers, analyzers, profilers, and others. Using these tools can help in optimization efforts or diagnose aspects of the project, code, or content that may not be working as expected.

You can access the developer tools from the Unreal Engine main menu bar under Window and Developer Tools.

The developer tools are divided into three categories:

- Debug—Contains debugging tools;
- Log—Contains the message log and output windows;
- Miscellaneous—Includes registrars, managers, profiles, and others.

### *3.7. Blueprint Debugging*

Blueprint Debugging is a very powerful feature that allows pausing the game in Play in Editor or Simulate in Editor mode and running through any Blueprint or Level Blueprint using breakpoints.

#### 3.7.1. Debugging Controls

The Blueprint debugger provides control over game execution during PIE and SIE sessions. The controls become available on the toolbar when the game is running. Different debug controls appear depending on the type of Blueprint being debugged and the current state of the debug session. Some controls only become active when needed, such as when a breakpoint is hit.

Checkpoints

Breakpoints are markers that can be placed on Blueprint graph nodes. When a node with a breakpoint is about to be executed in PIE or SIE mode, the game is paused, and the developer navigates to the node in the Blueprint editor's graph view. This makes it possible to observe the values of variables and explore or walk through the execution flow in the Blueprint. All breakpoints for a given Blueprint are displayed in the Debug tab and can be viewed in the Blueprint chart when selected. To place a breakpoint on a node, right-click the node and select Add Breakpoint from the context menu. A solid red octagon will appear in the upper left corner of the node. A breakpoint can be removed by right-clicking the node again or by right-clicking the breakpoint entry in the Debug tab and selecting the Remove Breakpoint command.

View window

The Blueprint Viewer is designed to speed up debugging by giving you access to the variables and nodes you want to monitor, even across multiple Blueprints. Observation data from each Blueprint you open in the editor that is part of the current call stack will be merged into a single list that will be populated with the current data whenever the Blueprint is paused. Using this list, you can view the variables and outputs of functions and easily switch between drawings. Arrays, sets, maps, and other data structures can be extended, making detailed inspection of any data they contain fast and convenient. You can also click on an entry in the "Node Name" column to navigate to a named node in any drawing, or select an entry in the "Object Name" column to select that particular object instance.

Call stack

The call stack available during debug sessions is similar in concept to the call stack used in most C++ development environments. The call stack shows the flow of execution between the Blueprint and native (C++) code, with the Blueprint function currently executing at the top of the stack.

When a breakpoint is hit, the call stack lists the functions that are currently executing, starting with the current function at the top and working its way down to the calling functions. This means that each line entry contains the name of the function that was called by the function listed on the line immediately below it. In the case of a recursive (self-calling) function, the same function name can appear multiple times in a row.

### 3.7.2. Widget Reflector

Widget Reflector is a tool that allows developers to define and debug Slate widgets.

The Unreal Editor UI was built using the Slate UI framework, and the Widget Reflector tool allows developers to define the Slate API used to display various widgets for the toolbox.

The widget reflector is built into the editor by default and is designed for developers looking to optimize and debug the project's user interface.

To open the Widget Reflector while the editor is running, select Window > Developer Tools > Widget Reflector. Alternatively, type Ctrl + Shift + W or type WidgetReflector into the console to open the tool.

Pick Hit-Testable widgets will track your mouse cursor and show the Slate hierarchy associated with the widget below it. If you click on the Source category to the right of the widget you want to see, you will be taken to the source code.

You may notice that some clickable resources refer to the actual Slate source code (SInlineEditableTextBlock.cpp, for example). Sometimes it can be suitable, but more often it is not desired to see what the code is actually calling, especially if one has not used Slate much before. To solve this problem, just keep going up the call stack in the reflector widget.

Widget Reflector has many good use cases. One useful aspect is to see how your game's user interface is being built at a more detailed level to help with optimization. It was also found that this tool is especially useful when extending the UE editor. This makes it quick and easy to find examples of Slate usage.

The classes discussed in the article are inherited from the base classes of Unreal Engine 5, on the basis of which the Unreal Engine 5 extension was developed. It needs to correspond to the Unreal Engine 5 class architecture and be able to work with new elements, for example, graph nodes, etc.

The tools developed in the project are presented as a plug-in—an independently compiled software module dynamically connected to the Unreal Engine game project to expand its functionality for virtual mines. The plugin consists of one project code module. The module class implements the functions of registering and canceling the main objects of the system in external modules.

Asset Utility System is a simple object that links the entire system because it combines the graph of executable nodes and the data of the board, which stores the external variables of the nodes. The Utility System asset editor is a descendant of FWorkflowCentricApplication, which has a Utility System application mode. Its tabbed functionality is used for behavioral graph development and the utility architecture. The Blackboard application mode is necessary for working with external variables in task nodes.

The Utility System node graphic system is part of the asset editor system that works with Unreal Engine entities used in various node editors, such as the Material Editor, which is used to develop the visual display of objects (mining workings and equipment, workers, etc.).

In addition to managing application modes, their tabs, and relationships, the Utility System asset editor is also the link for graphical and internal representations of the graph and its nodes.

The Graph Internal Representation Utility System is a class that derives from FAIGraphEditor because it provides basic functionality for working with nodes.

To execute the behavior pattern of an intelligent agent built on the Utility System architecture in the behavior tree, we developed a node that executes the behavioral graph of the Utility System. The Utility System node for the behavior tree uses the same execution procedure, but the behavioral graph is traversed only once.

The actual testing of the developed solutions was carried out at the DREAMVIAR LLC studio and formed the basis of the "HeadGun" project of the winner in the category "Best Sensorium VR Game" at the second nationwide game development competition at the "Unreal Engine Dev Contest"—https://youtube.be/tPt3XkwVc6A (accessed on 10 May 2023). The operability of the proposed solutions is beyond doubt, which made it possible to obtain a plug-in connected to the Unreal Engine 5 version of the game engine, which will speed up the creation of new simulators for virtual mines based on artificial intelligence.

The developed plug-in is of interest to developers and designers of gaming artificial intelligence tools that allow using the advanced method for designing virtual mines, the architecture of which is based on utility. The tool was originally used to develop artificial intelligence in a virtual reality game, and at that time the beta testing was approved by the developers.

## 4. Discussion

The results of the study of the possibilities of using the Unreal Engine game engine indicate its significant prospects for the virtual modeling of professional situations that the miners will face.

### *4.1. Utility System Plugin*

The developed project is presented as a plug-in—an independently compiled software module that is dynamically connected to the Unreal game engine project. The plugin is called Utility System Plugin (USPlugin).

#### 4.1.1. Utility System Module

The Utility System plugin consists of one module for storing code. Since the development of the module affects the work with the engine core, Slate input system, and application modes, the Utility System module has dependencies on the corresponding modules. The list of dependencies for this module is presented in the form of a table below (Table 1).

**Table 1.** Developed module dependency names (formed by Authors).

| Private Dependency Module Names | Names of Public Dependency Modules |
|---|---|
| ApplicationCore | Core |
| InputCore | CoreUObject |
| Slate | Engine |
| SlateCore | |
| EditorStyle | |
| UnrealEd | |
| GraphEditor | |
| KismetWidgets | |
| PropertyEditor | |
| AIGraph | |
| AIModule | |
| ToolMenus | |
| GameplayTasks | |
| GameplayTags | |

#### 4.1.2. Utility System Interface

The Utility System Module Interface (IUSPlugin) is a public interface that inherits from IModuleInterface, the base interface that all Unreal Engine modules must implement. The Utility System module interface provides common module methods for use by other classes. It also provides a static method for getting a module instance as a singleton-like access.

Provides access to methods:

- To create the Utility System asset editor;
- To access the cache;
- To access the asset category bitmask;
- To access the Stale style (FSlateStyleSet).

#### 4.1.3. Utility System Module Class

The Utility System module class is a hidden class that implements the methods of the IUSPlugin and IModuleInterface interfaces. IModuleInterface provides the StartupModule and ShutdownModule methods. ShutdownModule is called immediately after loading the module DLL and creating the module object. When StartupModule is called, this module will already have access to all other modules specified in the dependencies. ShutdownModule is called before the module is unloaded, just before the module object is destroyed. During normal shutdown, this is called in the reverse order that the modules exit StartupModule(). This means that as long as a module references dependent modules in StartupModule(), it can also safely reference those dependencies in ShutdownModule().

When the Utility System module is loaded, the FAssetTypeActions_UtilitySystem is registered with the AssetTools module. A Stale style is being created and brushes are being created, to access the .png images needed to be used as icons in the Utility System editor. The category bitmask is obtained via AssetTools. When the module is disabled, previous registrations are canceled, and the cache is cleared.

### *4.2. Asset Utility System*

A Utility System (UUtilitySystem) asset is a piece of Unreal Engine content represented as a simple object (UObject). The Utility System asset is the link between the entire system, as it combines the graph of executable nodes with the data of the board, which stores the external variables of the nodes. Moreover, it implements the IBlackboardAssetProvider interface.

IBlackboardAssetProvider is a helper interface for enabling FBlackboardKeySelector properties in DataAssets (and others). It is used by FBlackboardSelectorDetails to access the associated Blackboard based on the UObject hierarchy. The resource containing the Blackboard must broadcast OnBlackboardOwnerChanged when the asset ID changes.

### *4.3. Utility System Asset Editor*

#### 4.3.1. Factory Utility System

The Utility System Factory (UUtilitySystemFactory) inherits from UFactory and is responsible for creating UUtilitySystem instances. To communicate with an asset, the factory class constructor specifies the UUtilitySystem class as the supported class. Overrides the virtual method of the UFactory class responsible for creating the FactoryCreateNew instance, creating the instance using the NewObject<UUtilitySystem> template object creation method.

#### 4.3.2. Utility System Asset Type Actions

Utility System asset type actions (FAssetTypeActions_UtilitySystem) is a class inherited from FAssetTypeActions_Base. In the actions in the type, the category of the asset is defined, in which the context menu button for creating the asset will be located. Since the Utility System is one of the game AI design systems, it was located in the same section as the behavior tree system—“Artificial Intelligence”.

The actions on the Utility System asset type define the method to open the asset editor. In fact, this method determines which asset editor will be created and opened for the Utility System instance.

### 4.3.3. Utility System Asset Editor Application

The Utility System asset editor application (FUtilitySystemEditor) is the central class of the Utility System asset editor, which combines all the classes responsible for the graphical component of the entire system. The asset editor application is a multi-child of the following classes:

- FWorkflowCentricApplication is the base class for the asset editor that allows the asset editor to work like an application with multiple modes, having a specific set of tabs and a unique toolbar for each of the application's modes.
- FAIGraphEditor is a derived class, FEditorUndoClient, which is an interface for tools that want to handle undo/redo operations. FAIGraphEditor is the base class for working with a graph panel. It has a weak pointer to the Slate graph editor widget, as well as basic methods for working with graph nodes, such as copying, pasting, or deleting selected nodes.
- FNotifyHook is an interface whose methods are called when class properties change.

The main functions of the editor are grouping and initialization of internal entities, binding actions to commands and their execution, as well as updating interface elements in accordance with the selected application mode.

The Utility System asset editor consists of two application modes:

- Utility System mode—For direct work with the editor of the Utility System graph and with the parameters of the nodes of this graph.
- Blackboard mode—For working directly with the Blackboard key editor.

The tab structure of the Utility System asset editor is shown in Figure 13.

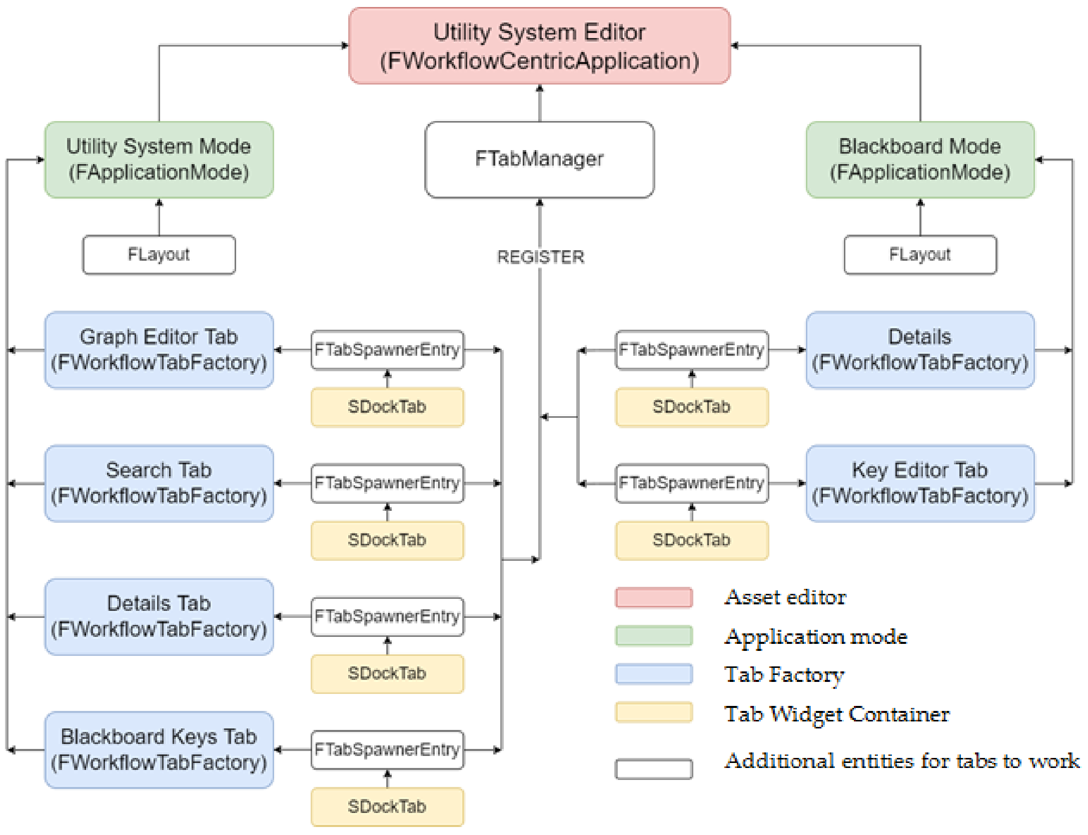

Utility System Editor (FWorkflowCentricApplication)
Utility System Mode (FApplicationMode)
FTabManager
Blackboard Mode (FApplicationMode)
FLayout
REGISTER
FLayout
Graph Editor Tab (FWorkflowTabFactory)
FTabSpawnerEntry
SDockTab
FTabSpawnerEntry
SDockTab
Details (FWorkflowTabFactory)
Search Tab (FWorkflowTabFactory)
FTabSpawnerEntry
SDockTab
FTabSpawnerEntry
SDockTab
Key Editor Tab (FWorkflowTabFactory)
Details Tab (FWorkflowTabFactory)
FTabSpawnerEntry
SDockTab
Blackboard Keys Tab (FWorkflowTabFactory)
FTabSpawnerEntry
SDockTab
Asset editor
Application mode
Tab Factory
Tab Widget Container
Additional entities for tabs to work

**Figure 13.** Utility System asset editor tab structure.

#### 4.3.4. Utility System Toolbar Builder

The Utility System toolbar builder is a class that works with asset editor menu expanders. The toolbar builder uses the ToolbarExtenders array and other menu extension entities from the Utility System asset editor parent class, FAssetEditorToolkit, to make changes to the menu state.

The toolbar builder implements methods for adding application mode switching buttons and binding appropriate handlers to them.

To switch between application modes, each of the application modes adds widgets with mode buttons when expanding the toolbar. To separate the mode switches from other actions, when adding them, the toolbar is expanded with widgets since the horizontal container for widgets is to the right of the container for widgets generated by expanders.

### *4.4. Utility System Application Mode*

The Utility System application mode (FApplicationMode) provides a set of tabs required for working with the utility-based game AI design graph. The Utility System application mode expands the asset editor toolbar and expands the main menu by adding buttons to the "Window" section to control the display of the tabs added by the mode.

#### 4.4.1. Extending the Toolbar

The toolbar extension happens in the constructor of the application mode class. To extend the asset editor toolbar, application mode uses a shared pointer to an instance of the Utility System asset editor.

The Utility System mode adds the following extensions to the toolbar to create new assets that are involved in the design of game AI:

- New Task—The action of creating a new asset of the UUSTask_BlueprintBase class.
- New Factor—The action of creating a new asset of the UUSFactor class.
- New Blackboard—The action of creating a new asset of the UBlackboardData class.

Although each action is the creation of a new asset, the algorithms for its creation are different. The reason is that to be created correctly, objects that have their own asset editor must be created using that asset's creation factory. Moreover, when creating a new Blackboard, additional rules are applied to it—this is the initialization of the new Blackboard in the properties of the Utility System asset editor and in the properties of the root node of the graph.

#### 4.4.2. Graph Editor Tabs

To spawn tabs, the Utility System application mode uses special factories for each table. Each tab factory implements methods for displaying the name and description of the tab, as well as methods for creating a Slate widget for the body of each tab.

Graph panel tab

The graph panel tab factory uses the Slate widget of the SGraphEditor class as the tab body. The tab widget is initialized through the asset editor. Moreover, when initializing the widget, the asset editor binds context menu commands to the graph editor panel.

Detail tab

To create a details widget, an object that implements the IDetailsView interface is used. To create this object, use the CreateDetailView static method of the FPropertyEditorModule property editor module. The created DetailsView object is used to spawn a widget that displays the properties of the object that is specified to be displayed. The DetailsView shows the properties of the selected node in the graph pane; if no node has been selected, then by default the properties of the root node are shown.

Search tab

The graph panel tab factory uses a Slate widget of the SFindInUS class as the body of the table. SFindInUS uses the Utility System asset editor as an initialization object to search by its properties. The search tab is used to search through graph nodes. The search occurs when the user enters text characters into the search string. Each time a character is entered, the nodes of the Utility System graph are searched for, including the final substring as part of the node name. For the found nodes, a map is compiled linking the pointers to the nodes and their names. Slate widgets of text strings are then created from the resulting list and placed in a vertical container of search result items. For each element, click event processing is added, which focuses the graph panel on the node selected by name.

Whiteboard keys tab

The graph panel tab factory uses a Slate widget of the SUtilitySystemBlackboardView class as the tab body. The tab widget is initialized through the asset editor. When the widget is initialized, the asset editor passes a pointer to the existing Blackboard to the widget. When you change the Blackboard, the pointer in the widget is also updated, and the content of the widget is rebuilt. The board keys widget uses its properties to generate content. The board key widget only shows what properties exist in the Blackboard object and does not have any handling for changing them.

Utility System tab layout

The layout of the Utility System mode tabs is performed in the Utility System application mode class constructor using the tab manager object. Tab layout is performed by specifying tab containers, specifying the orientation for nested components (horizontal or vertical), and populating these containers with tabs or nested containers.

## *4.5. Blackboard Application Mode*

The board holds memories for an individual agent or agents. In Unreal Engine, boards work together with a behavior tree. This provides easy and direct access to variables from any of the nodes in the behavior tree. Similar to the behavior tree, the Utility System has a graph representation and also uses boards to store the external variables of task nodes or any other nodes.

Board variables are stored using key-value pairs that are defined for each specific Blackboard type. Each entry in Blackboard consists of a "key" string and a value of a specific type. Blackboard key values support Unreal Engine primitives and object types. For object types, you can set the base class.

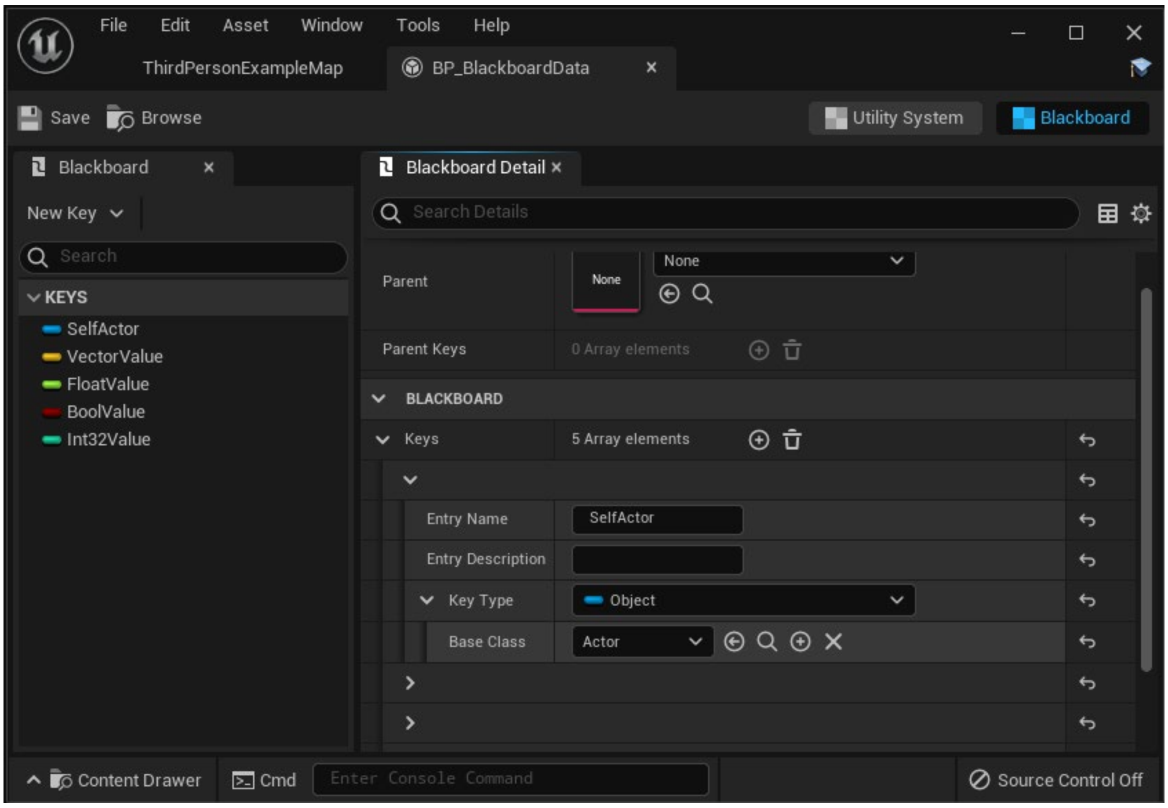
It can be seen from the above that the Blackboard application mode is already implemented in the behavior tree asset editor system. However, one cannot just take and add an already existing Blackboard mode as an application mode for the Utility System asset editor since the existing application mode is designed to work directly with the behavior tree asset editor data types. Based on this, a similar application mode was developed, but for working with Utility System data types.

### 4.5.1. Blackboard Tabs

The Blackboard application mode consists of two tabs: a tab for displaying a list of keys and a detail tab for editing keys and their settings. The board keys tab is exactly the same tab for displaying keys as the tab described above. The Blackboard Details tab also consists of an object that implements the IDetailsView interface, which uses the Blackboard data asset to display and modify Blackboard properties.

### 4.5.2. Blackboard Tab Layout

The layout of the Blackboard mode tabs is performed in the Blackboard application mode class constructor using the tab manager object. Tab layout is accomplished by adding the tabs described above to a horizontal container. The result of the Blackboard application mode tab layout is shown in Figure 14.



KEYS
SelfActor
VectorValue
FloatValue
BoolValue
Int32Value

Parent: None
Parent Keys: 0 Array elements
BLACKBOARD
Keys: 5 Array elements
Entry Name: SelfActor
Entry Description:
Key Type: Object
Base Class: Actor

**Figure 14.** Blackboard tab layout.

### *4.6. Graphical Node System Utility System*

The Utility System Node Graphics System is a system created for designing game AI using a Utility System approach. The graph node system allows developers and designers of artificial intelligence games to work with design entities in the form of graph nodes. The development of game artificial intelligence using the graphical system of nodes of the Utility System occurs by building a behavioral graph using nodes such as task nodes and special choice nodes.

A task node is a graph representation for a specific game AI task asset. A task asset is an element of a behavioral graph design and execution system that provides game AI developers and designers with methods to determine the sequence of any steps or actions to be performed by an AI agent. In other words, the AI task node is a separate AI agent behavior algorithm, represented as a graphic graph node. Typically, actions represented by task nodes are atomic operations on an agent that are controlled by the AI system. Used as primitive operations, for example: enabling or disabling animation for an agent's skeletal mesh, playing an audio effect, or creating a particle system. So are more complex ones, such as searching for a point on the map that satisfies some conditions, building an optimal route to the found point, and moving the agent along this route to this point.

A special choice node (useful choice node) is a graph representation of a special asset that uses utility evaluation approaches to determine the actions that are most "useful" during the evaluation phase. A useful choice node is a node that can have multiple graph outputs. The useful choice node is extended with special evaluation subnodes. Each expansion by an evaluation subnode adds a new graph output. Thus, the useful choice node is a kind of hub that, when executed, collects numerical data on utility estimates from subnodes, finds the subnode with the highest utility, and gives the execution system an output opposite this subnode. In this way, a useful selection of subsequent nodes of the behavioral graph for execution is made. A graphical representation of the selection nodes and tasks is shown in Figure 15.

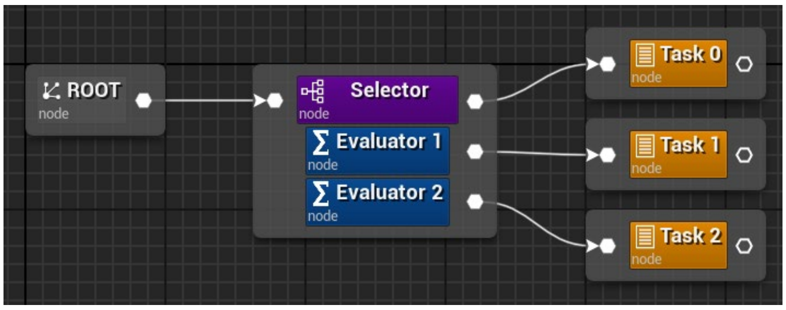


**Figure 15.** Graphical representation of select and task nodes.

### 4.6.1. General Structure of the Graphic System of Nodes

The Utility System node graphics system is part of the asset editor system that works with Unreal Engine entities used in various node editors, such as the Blueprint editor, which is used for graphical programming, or the material editor, which is used to develop the visual display of objects. The layout of the graphic system of nodes is shown in Figure 16.

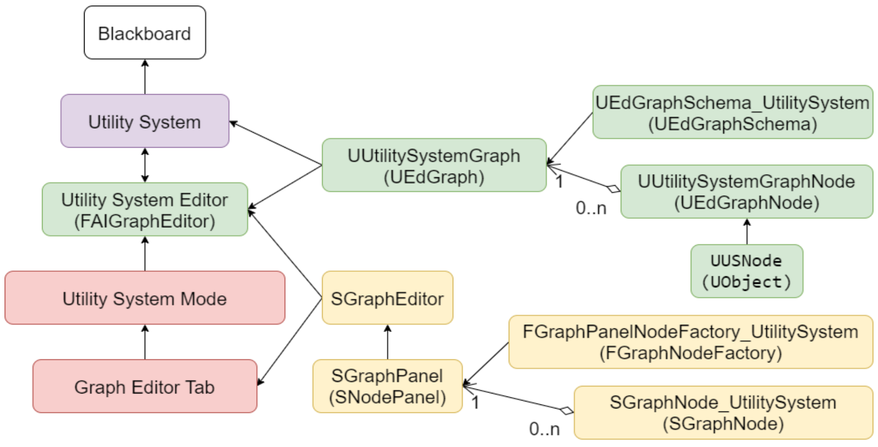

Blackboard
Utility System
Utility System Editor (FAIGraphEditor)
Utility System Mode
Graph Editor Tab
UUtilitySystemGraph (UEdGraph)
UEdGraphSchema_UtilitySystem (UEdGraphSchema)
UUtilitySystemGraphNode (UEdGraphNode)
1
0..n
UUSNode (UObject)
SGraphEditor
SGraphPanel (SNodePanel)
FGraphPanelNodeFactory_UtilitySystem (FGraphNodeFactory)
SGraphNode_UtilitySystem (SGraphNode)
1
0..n

**Figure 16.** Scheme of the device of the graphic system of nodes.

In the diagram above, the main entities of the graphic system of nodes and their relationships can be seen. The whole system can be divided into the following components:

- Data storage assets.
- Blackboard—An asset that stores the external variables of the nodes.
- Utility System—The asset for which the editor is designed; used to store graph data and its nodes and for linking nodes to Blackboard keys.
- Entities that work with the internal representation of a graph:
- FUtilitySystemEditor—As part of the graphical node system, the Utility System asset editor is the connecting link between the internal representation of the graph and its nodes and the graphical representation of the graph panel and the graphical representation of the nodes.
- UUtilitySystemGraph—A key entity that is responsible for loading, updating, and operations with graph nodes.
- UEdGraphSchema_UtilitySystem—An entity that works with various actions applicable to the graph and its nodes. For example, the connection of nodal connectors or actions of context menus.
- UEdGraphNode—The entity of the graph node as part of the graph.
- UUSNode—Node instance.
- Entities that work with the graphical representation of the graph:
- SGraphEditor—Wrapper interface for Graph Editor Slate widgets.
- SGraphPanel—Slate-panel, for ordering child Slate-widgets.

- SGraphNode—Slate widget that is a graphical representation of a graph node
- FGraphPanelNodeFactory_UtilitySystem—Node graphical representation generation factory.
- Entities that the asset editor that spawns the graphical tabs of the asset editor:
- Utility System Mode—Application mode that creates asset editor tabs.
- Graph Editor Tab—An asset editor tab that wraps the graph editor interface.

#### 4.6.2. The Utility System Asset Editor as a Link

In addition to managing application modes, their tabs, and their relationships, the Utility System asset editor is also the link for graphical and internal representations of the graph and its nodes. To work with the graphical representation of a graph, the Utility System asset editor extends the FAIGraphEditor class by inheritance. FAIGraphEditor already contains a weak pointer to a wrapper interface for the Graph Editor Slate widget and some methods to work with it. FAIGraphEditor implements the following commands applicable to the internal representation of graph nodes (UEdGraphNode): Select all nodes; Delete selected nodes; Copy selected nodes; Cut and Paste selected nodes; Duplicate nodes; Cancel; Redo.

When you first set the Utility System application mode for an asset editor, the asset editor creates a new graph and graph diagram and links them. The asset editor then initializes the generated graph, creates a root node for it, and then links the generated graph to the Blackboard.

The asset editor performs the function of associating graph nodes with tabs in the Utility System application mode. When adding new graph nodes through the asset editor, the graph nodes report their changes to their fields and whether a particular node is selected. The asset editor processes these messages and updates the details tab.

#### 4.6.3. Utility System Graph

The base class of graphs in Unreal Engine is the UEdGraph class, which has many derived classes, for example: UAnimationGraph, UMaterialGraph, etc., which are used to implement the functions of various modules. The UAIGraph class, which is derived from UEdGraph, was chosen as the main class for the Utility System graph because it has extended basic functionality for working with nodes and also because the methods of this class are used in FAIGraphEditor. Utility System Graph (UUtilitySystemGraph).

#### 4.6.4. Chart Schema Utility System

A graph schema is an entity that deals with various actions applicable to a graph and its nodes. The graph diagram of the Utility System has the following functions:

- Creates default graph nodes (root node);
- Extends the actions of the graph context menu;
- Expands node context menu actions;
- Checks the connection validity of graph node connectors;
- Controls connection colors;
- Creates connection drawing policy;
- Handles cache operations.

The graph diagram extends the actions of the graph context menu in such a way that, through the context menu, one can add graph nodes to the graph panel as well as perform other actions applicable to the graph. Through the context menu, nodes belonging to certain classes can be added to the graph panel. Actions are extended using the context menu builder (FGraphContextMenuBuilder). For the context menu builder, the available assets for each category of node classes (task nodes and composite nodes) are retrieved from the class cache. The context menu is rebuilt each time it is called. Thus, all created nodes appear as actions in the context menu when it is reopened. The graph context menu is shown in Figure 17.

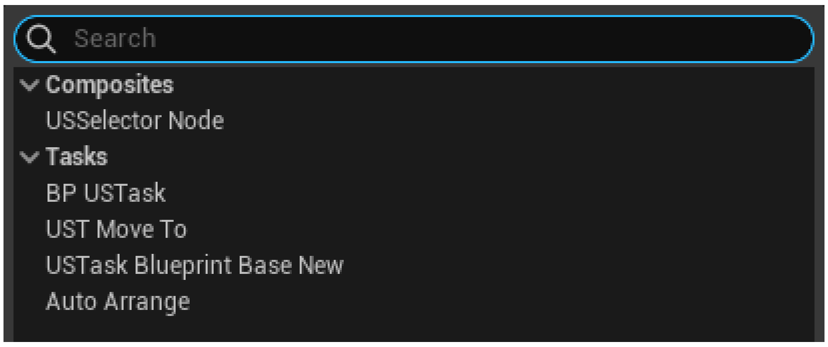
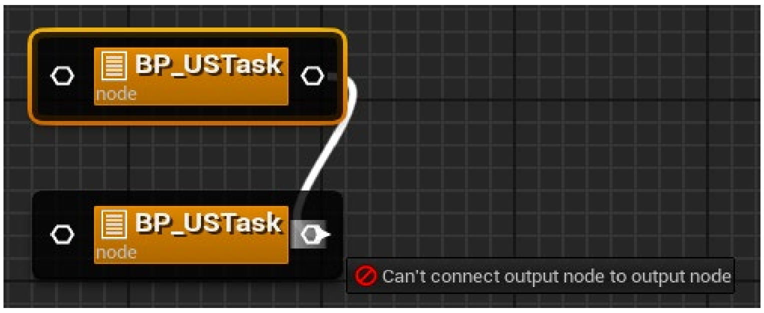


**Figure 17.** Graph context menu.

Additional context menu actions that are not related to adding new nodes are classes derived from FEdGraphSchemaAction. Graph actions must define a PerformAction method that can be called when an action is selected from the context menu.

Auto Arrange is an action that performs automatic ordering of graph nodes. The methods of the standard ordering library (USAutoArrangeHelpers) are used to arrange nodes.

Connecting graph nodes is performed by dragging a connector from an output connection point of a graph node to an input connection point of another graph node. Connecting two connection points is allowed if the connection points are opposite each other (input and output), the connection points must not belong to the same node, and the graph obtained by the connection must not be looped. The handling of an invalid connection is shown in Figure 18.



Can't connect output node to output node

**Figure 18.** Handling an invalid connection.

In order to handle the admissibility of connecting two connection points in the class of a graph diagram generated from UEdGraphSchema, it is necessary to define the CanCreateConnection method, which returns an FPinConnectionResponse connection response structure. The structure contains the text of the message and an enumeration of type ECanCreateConnectionResponse with values that allow and do not allow connections.

### 4.6.5. Connection Drawing Policy

A connection drawing policy is a class that draws connections for a UEdGraph composed of pins and nodes. The connector drawing policy is responsible for the appearance of the line that will be visible to the user when connecting two nodes. It is possible to define whether the connection line of the nodes will have only a straight direction or have bends like a Bezier curve. One can also define the image at the ends of the connecting line, for example, whether it will be arrows or any other element.

To develop your own connection drawing policy, you need to create a class derived from FConnectionDrawingPolicy and define connection drawing methods for it. In order for the developed policy to be applied, it is necessary that the diagram graph create an instance of this policy and return it as a result of the CreateConnectionDrawingPolicy method.

FConnectionDrawingPolicy provides various methods for drawing connections, lines, and calculating their tangents, so in most cases it is enough to use the FConnectionDrawingPolicy methods. The connection rendering policy connection line is shown in Figure 19.

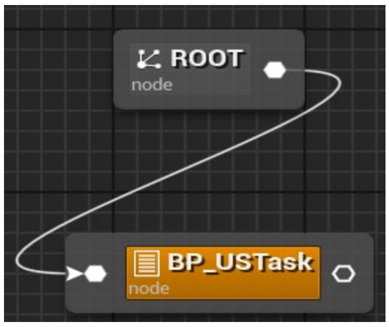


**Figure 19.** Connection rendering policy trunk.

The Utility System graph connection drawing policy draws a connection curve with a static arrow at the end of the connection that indicates the direction in which the graph should be executed.

### 4.6.6. Development of System Nodes

A graph node can be divided into several entities, each of which exists at its own graph representation level. There are three graph representations:

- Visual representation of the graph
- Representation of a graph as a set of objects with connections
- Representation of the graph as a runtime system

Accordingly, for each representation level, the representation of the node in this graph must be defined.

Base Graph Node

The base Graph Node Utility System (UUtilitySystemGraphNode) is a class derived from UAIGraphNode. UAIGraphNode extends the base class of all graph nodes (UEdGraphNode) by adding several methods to be able to work with the level system of nodes, in which child nodes can be part of parent ones. However, the presence or absence of subnodes is not registered in the visual representation. Therefore, when developing a Slate component, it is necessary to take into account the presence of an additional Slate container for subnodes and methods that update the visual component when subnodes are added or removed.

The base node of the Utility System graph defines methods for adding node connectors and initiates default values for methods that will affect the visual component:

- Host headers
- Images that label knots
- Node colors

The base node also adds context menu extension methods for adding child nodes. Nodes inherited from the base will use these methods to construct a context menu, determining exactly what types of child nodes are allowed for a given node type.

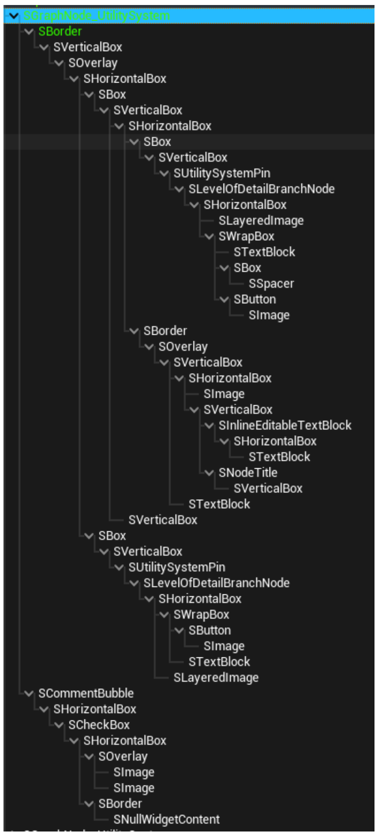
The visual component of a graph node

The visual component of the graph node is the Slate widget, which is responsible for the appearance of the nodes and their connectors. The visual component of a graph node consists of a class derived from SGraphNode that defines the appearance of the node and a class derived from SGraphPin that defines the appearance of connectors.

The visual component of the graph has a pointer to the graph node (UEdGraphNode), so usually the visual component of the graph node is designed in such a way as to use the values of the graph node and change its appearance in accordance with them. For example, use the title, icon, color, etc. from a graph node to generate the corresponding Slate element according to its UEdGraphNode element and not to create a unique Slate widget for each node type.

Designing a graph node Slate widget is a container layout of Slate elements, specifying different parameters for each element (padding, brushes, alignment types, etc.), as well as binding events for active elements.

The general structure of the developed tree of Slate elements can be seen in the image from the widget reflector (Figure 20).



```
SGraphNode_UtilitySystem
  SBorder
    SVerticalBox
      SOverlay
        SHorizontalBox
          SBox
            SVerticalBox
              SHorizontalBox
                SBox
                  SVerticalBox
                    SUtilitySystemPin
                      SLevelOfDetailBranchNode
                        SHorizontalBox
                          SLayeredImage
                          SWrapBox
                            STextBlock
                            SBox
                              SSpacer
                            SButton
                              SImage
                SBorder
                  SOverlay
                    SVerticalBox
                      SHorizontalBox
                        SImage
                        SVerticalBox
                          SInlineEditableTextBlock
                            SHorizontalBox
                              STextBlock
                          SNodeTitle
                            SVerticalBox
                      STextBlock
              SVerticalBox
          SBox
            SVerticalBox
              SUtilitySystemPin
                SLevelOfDetailBranchNode
                  SHorizontalBox
                    SWrapBox
                      SButton
                        SImage
                      STextBlock
                    SLayeredImage
  SCommentBubble
    SHorizontalBox
      SCheckBox
        SHorizontalBox
          SOverlay
            SImage
            SImage
          SBorder
            SNullWidgetContent
```

**Figure 20.** Slate tree of node elements.

The structure of the Slate elements of the node is shown in Figure 21.

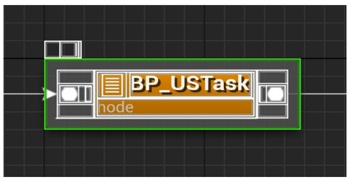


**Figure 21.** The structure of the Slate elements of a node.

The visual component of the graph node handles the following events that interact with the graph node:

- Move;
- Double-click mouse;
- Mouse hover;
- Clicking the right mouse button;
- Connecting nodes.

When the mouse cursor is hovered, the visual component of the graph node creates a Slate element of the text description of the node and adds it to the graph panel next to the mouse cursor; when the mouse cursor is moved away, the text description element is destroyed.

Right-clicking opens the context menu of the node menu.

When connecting and disconnecting nodes, the connection graph is searched to find the root node. If the root node is connected to the current one, then it is considered active and changes the color of the outer frame. When the node is disconnected, the check is repeated, and if the node is no longer connected to the root, then it becomes inactive and the color of the frame also changes. The selection of the contour of connected active and inactive nodes is shown in Figure 22.

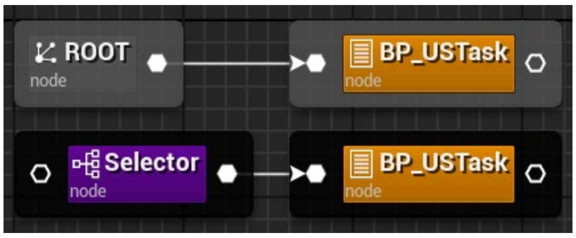


**Figure 22.** Highlighting the contour of connected active and inactive nodes.

Base node instance

The base node instance is a simple Unreal Engine object that implements the IGameplay-TaskOwnerInterface interface. The base node instance also contains initialization methods.

IGameplayTaskOwnerInterface interface methods:

- Get the "body" of the task owner/default having world location;
- Get default priority for task execution;
- Get the owner of the task, or default if the task is invalid;
- Find task component;
- Notification is triggered after the state changes to active;
- Notification is triggered after the status changes to completed or paused;
- Notification is called after the completion of initialization.

Reference node

The reference node, or root node (Root node), is the node of the system from which the traversal of the graph begins during the Utility System when it is executed. All nodes that do not have a connection to the root node are considered inactive and are not allowed to run. The root node is the only node that has a pointer to the system's Blackboard. In fact, all nodes that interact with Blackboard receive it through the root node. The system also accesses Blackboard only when it is installed in the root node. The image of the root node is shown in Figure 23.

**Figure 23.** Root node.

The root node has a single output connector and no input connectors since the graph starts at it. Moreover, the root node has an empty node instance since it is only a starting point in the execution of the graph and does not contain any actions.

Task node

A task node instance is an abstract game AI task class. It provides the methods needed to be overloaded in the derived class, which are called when the execution of the graph reaches this node.

Methods selected for overriding are:

- Called when the execution reaches the task;
- Tick method that is called every game frame until the task is completed;

- The method that is called when the task is completed from outside.

The execution methods you define provide the controller and pawn of the game AI agent as arguments. Thus, it will be more convenient for an AI developer who will design an action algorithm for a specific task to control the AI agent. In order for the runtime system to understand that the active task has completed, the developer must call the task's completion method, indicating whether it completed successfully or failed.

Useful selection node

The useful choice node is used by the runtime system to determine how the nodes will be executed. A useful choice node is the only node for which child subnodes are allowed. The child subnodes of the useful choice node are the evaluator nodes. Since the mechanism of a useful choice node will always work according to the same principle and, if necessary, be extended with the help of child subnodes, the node class is declared complete, which makes it impossible to derive classes from it. The principle of execution of the useful choice node is based on the work of its child nodes—evaluators.

Utility evaluator node

The utility evaluator node is also a complete node that does not allow further inheritance. Utility evaluators are needed so that game AI agents can make "weighted decisions" before taking any action. The key role of the utility evaluator is to collect the final numerical assessment of the useful contribution of the actions that will be performed at the next step of the graph execution.

The assessment of the usefulness of actions is a calculus consisting of many factors that can influence these actions.

A factor is a special entity whose purpose is to obtain a numerical value for some physical relationship in the game space. These can be either directly calculated values of physical quantities, for example, the distance from the intelligent agent to the key points on the map that the agent must hit, or more abstract concepts represented by some categorical quantities, for example, will the intelligent agent be detected by the enemy if it moves to the key point (discovered/hidden)?

A factor is a normalized absolute numerical indicator of some relation in the game space. This means that for the same ratios, the factors must calculate the same numerical indicator. To assess the beneficial effect of this factor on actions, the numerical indicator of the factor is additionally processed using utility curves.

A utility curve is a utility function for a factor that is applied to a factor when estimating its utility for actions that follow a particular estimator. For example, if the estimator is mapped to the action of moving to the nearest keypoint, then applying a decreasing function to the distance factor will have the greatest utility for nearby keypoints and the least for distant ones.

From the developer's point of view, a factor is an asset of the UUSFactor class, for which it is necessary to create a derived class and define its Evaluate method so that the factor returns the required numerical indicator. Additionally, a utility curve is a regular FFloatCurve curve object provided by Unreal Engine. The appraiser's node has an associative array that combines pairs of factors and their corresponding utility curve objects.

### *4.7. Utility System*

The Utility System graph execution system is a system that uses the developed behavioral graph Utility System to control an intelligent agent in the game world.

#### 4.7.1. Utility System Component

The Utility System component is a UActorComponent that needs to be added to the AI Controller component tree. The AI component contains a pointer to the Utility System asset, which must be specified for each AI controller. When a pawn of an intelligent agent appears in the game space, the pawn generates an AI controller for itself, containing the Utility System component. The AI controller uses the Utility System component to start the behavioral graph execution process.

### 4.7.2. AI System

An artificial intelligence system (UAISystem) is a system that links the AI component of a particular intelligent agent to the global AI manager and also links the AI component to the game environment query and navigation systems. The AI System for the Utility System extends the base one by adding interaction with the Global AI Manager for the Utility System.

### 4.7.3. AI Manager

The AI manager is a global gamespace object that handles registering behavioral graph patterns and instantiating nodes. The AI manager is spawned by the AI system in the game space for which the system was set. The AI component uses a manager to load patterns from the behavioral graph and its nodes. This is necessary so that many intelligent agents using the same behavior model do not create many identical, expensive behavioral graphs but instead use a single instance for each intelligent agent.

### 4.7.4. Task Node Execution

To execute a node of a task, the runtime system creates an instance object of that node, subscribes to the completion of task execution, and then initializes the object as active. After activating the task object, the runtime system enters waiting mode for the completion of the task. The task is completed only if the FinishExecute method was called during its execution, which will accordingly notify the system of the completion of the task, and the system can proceed to the execution of the next node.

### 4.7.5. Execution of the Useful Path Selection Node

Since the selection node is not a full-fledged executable entity but only a link for determining the next node, its execution is as follows.

If the select node has no estimators, then it will not perform any calculations and will simply pass the next node to the system. If a utility path selection node has utility estimators, then each estimator calculates its utility using pairs of factors and utility curves. After calculating all the estimates, the selection node determines which of the estimators has the maximum utility value and passes to the execution system the node, the transition to which occurs from the estimator subnode, with the maximum utility.

The process of calculating utility estimates is shown in Figure 24.

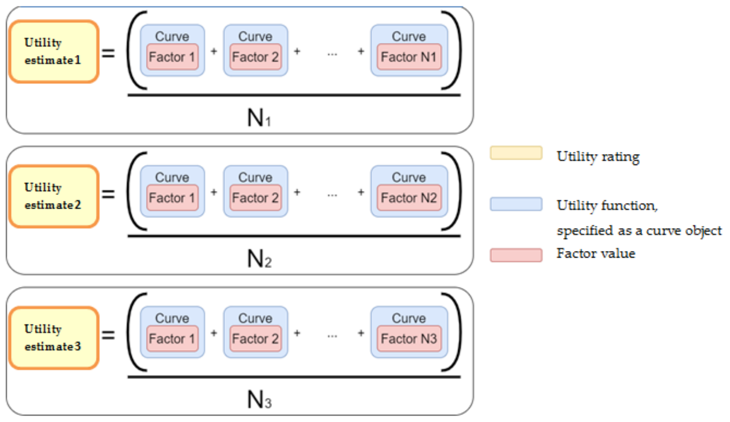


**Figure 24.** Computing utility estimates.

The graph execution system uses the utility evaluation subnodes located in the choice node to determine the subnode with the maximum utility and to navigate the connector of this subnode to the next execution node. Each utility evaluator uses the factors assigned to it to obtain numerical values for the factors. Then the numerical assessment of each factor is transferred to the utility function, represented as a curve object.

To execute the behavior pattern of an intelligent agent built on the Utility System architecture in the behavior tree, a node for the behavior tree was developed that executes the behavioral graph of the Utility System. The Utility System node for the behavior tree uses the same execution engine, but the behavioral graph is traversed only once.

#### 4.7.6. Utility System Launch Node for Behavior Tree

In order to be able to use the behavior pattern of an intelligent agent built on the Utility System architecture in the behavior tree system, a node for the behavior tree was developed that executes the behavioral graph of the Utility System. The Utility System node for the behavior tree uses the same mechanism; however, unlike the runtime Utility System, the node for the behavior tree executes the behavior graph without looping. Thus, after a single pass through the behavioral graph of the Utility System, the execution of the nodes of the behavior tree occurs in the usual order.

As a result of work on the practical part, a plug-in was developed that connects to the Unreal Engine 5 version of the game engine. The developed plug-in provides developers and designers of game AI with tools that allow the use of an advanced method for designing game AI, the architecture of which is based on utility.

The developed plug-in proposes to use a system for constructing a behavioral graph using nodes that evaluate the usefulness of actions to implement a complex branching system when choosing which of the chains of actions should be executed. This approach makes it possible to qualitatively create the illusion of awareness of the choice by an intelligent agent. The use of useful action selection nodes qualitatively affects the breadth of perception of the game space by an intelligent agent, since during the development process it is possible to use a variety of action evaluators, which will allow achieving reasonable behavior for an intelligent agent regardless of the variation of the game space parameters. Moreover, since a behavioral graph can be composed of several useful action selection links, game AI developers and designers are able to achieve the complex and deep behavior of an intelligent agent, which provides users with the most valuable experience of interacting with intelligent agents.

## 5. Conclusions

The importance of the development of gamification in the training of future and real miners based on virtual reality requires a direction for improving the software game engine. In the course of the work, the relevance of the topic was substantiated, associated with the practical need for tools for developing game artificial intelligence for the Unreal Engine game engine for use in the development of utility-based AI architecture.

The paper describes the main approaches to the development of game AI with a description of their advantages and disadvantages, as well as the differences between the development of game AI and academic AI. The paper described the key principles of the theory of utility, which is a key link for the development architecture of game AI based on utility.

The main approaches to the implementation of software tools were described. The Unreal Engine tools used to develop the interface component of the editor, as well as engine debugging tools and a tool for viewing and obtaining additional information about the Unreal Engine editor widgets, are considered. In the work, the architecture of the structure of the windows of the Unreal Engine editor and the methods of extension of toolbars and menus were disassembled.

The following tasks were successfully solved in this work:

- Implementation of the necessary components to apply a flexible approach to the development of game artificial intelligence Utility System;
- Development of the node editor for the Utility System asset;
- Development of basic computing nodes Utility System and task nodes;
- Development of the Utility System asset execution system;
- Integration of the runtime Utility System into the behavior tree.

Therefore, the main effect is saving time and labor costs for application development for Virtual Mine gamification, so we hope that this approach is interesting for programmers.

**Author Contributions:** Conceptualization, F.A.-A. and S.Z.; methodology, F.A.-A. and S.Z.; software, F.A.-A.; validation, S.Z.; formal analysis, S.Z.; investigation, F.A.-A.; data curation, F.A.-A.; writing—original draft preparation, S.Z.; writing—review and editing, S.Z.; supervision, F.A.-A. and S.Z. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Acknowledgments:** The authors acknowledge DREAMVIAR LLC for administrative and technical support and materials used for experiments. Special thanks to Denis Mazur and Nikita Volkov. The work is dedicated to Nikolai Borisov, a permanent teacher who made a significant contribution to the development of the programming school, which trained a whole generation of IT specialists.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Appendix A

List of terms:

A game engine—Software that allows you to create and run video games. It gives developers the tools to create most of the game's components and then lets them put them together.

Game artificial intelligence (AI)—A set of software techniques that are used in computer games to create the illusion of intelligence in the behavior of computer-controlled characters.

AI Task—A program script (a specific action) to be executed by an intelligent agent in the world.

Flexible game AI—In this case, the concept of flexibility should be understood as the absence of hard coding of the choice of AI execution scenarios or chains of these scenarios.

An AI agent (Intelligent Agent/Bot)—Any object that can perceive its environment and take actions using actuators.

World—In UE, this is the top-level object that represents the map in which Actors and Components will exist and be rendered.

Actor—In UE, this is the base class for an object that can be placed or instantiated in a layer.

Component—In UE, this is the base class for components that define reusable behavior that can be added to various types of Actors.

Pawn—In UE, this is the base class of all actors that can be owned by players or AI.

Controller—In UE, this is not a physical entity that can possess a Pawn to control its actions.

Asset—Part of the content for the Unreal Engine project; can be considered as a UObject serialized to a file.

A data asset—The base class for a simple asset that contains data.

Plugin—Independently compiled software module that is dynamically connected to the main program and designed to extend and/or use its capabilities. Plugins are usually implemented as shared libraries.

Widget—A small element of the graphical interface.

## References

1. Stothard, P. The Feasibility of Applying Virtual Reality Simulation to the Coal Mining Operations. In Proceedings of the AusIMM Proceedings: Mining Risk Management, AusIMM, Sydney, Australia, 9–12 September 2003; pp. 311–316.
2. Smith, G.L.; Con, C.; Gil, C. Virtual Mine Technology. In Proceedings of the Bowen Basin, Symposium, Rockhampton, Australia, 22–24 October 2000; pp. 1–26. [CrossRef]

3. Kim, H.; Choi, Y. Performance Comparison of User Interface Devices for Controlling Mining Software in Virtual Reality Environments. *Appl. Sci.* **2019**, *9*, 2584. [CrossRef]
4. Zhang, C.; Wang, X.; Fang, S.; Shi, X. Construction and Application of VR-AR Teaching System in Coal-Based Energy Education. *Sustainability* **2022**, *14*, 16033. [CrossRef]
5. Hu, T. Technological core and economic aspects of modernization on the Industry 4.0 platform. *Econ. Innov. Manag.* **2022**, *3*, 5–18. [CrossRef]
6. Carbonell-Carrera, C.; Saorin, J.L.; Melián Díaz, D. User VR Experience and Motivation Study in an Immersive 3D Geovisualization Environment Using a Game Engine for Landscape Design Teaching. *Land* **2021**, *10*, 492. [CrossRef]
7. Volkova, A.L.; Gasanov, M.A. The quadruple helix in the system of human-oriented structural transformation of the economy. *Econ. Innov. Manag.* **2021**, *2*, 4–12. [CrossRef]
8. Liarokapis, F.; Macan, L.; Malone, G.; Rebolledo-Mendez, G.; De Freitas, S. A pervasive augmented reality serious game. In Proceedings of the 2009 Conference in Games and Virtual Worlds for Serious Applications, VS-GAMES, Coventry, UK, 23–24 March 2009; pp. 148–155.
9. Yunqiang, C. An overview of augmented reality technology. *J. Phys. Conf. Ser.* **2019**, *1237*, 022082. [CrossRef]
10. Gandolfi, E. Virtual reality and augmented reality. In *Handbook of Research on K-12 Online and Blended Learning*; Carnegie Mellon University: Pittsburgh, PA, USA, 2018; pp. 545–561.
11. Danielsson, O.; Holm, M.; Syberfeldt, A. Evaluation Framework for Augmented Reality Smart Glasses as Assembly Operator Support: Case Study of Tool Implementation. *IEEE Access* **2021**, *9*, 104904–104914. [CrossRef]
12. Yang, C.; Tu, X.; Autiosalo, J.; Ala-Laurinaho, R.; Mattila, J.; Salminen, P.; Tammi, K. Extended Reality Application Framework for a Digital-Twin-Based Smart Crane. *Appl. Sci.* **2022**, *12*, 6030. [CrossRef]
13. Jalo, H.; Pirkkalainen, H.; Torro, O. Extended reality technologies in small and medium-sized European industrial companies: Level of awareness, diffusion and enablers of adoption. *Virtual Real.* **2022**, *26*, 1745–1761. [CrossRef]
14. Bauer, R.D.; Agati, S.S.; dá Silva Hounsell, M.; da Silva, A.T. Manual PCB assembly using Augmented Reality towards Total Quality. In Proceedings of the 22nd Symposium on Virtual and Augmented Reality (SVR), Porto de Galinhas, Brazil, 7–10 November 2020; pp. 189–198.
15. Salta, A.; Prada, R.; Melo, F.S. A Game AI Competition to Foster Collaborative AI Research and Development. *IEEE Trans. Games* **2021**, *13*, 398–409. [CrossRef]
16. Romero, D.; Sánchez, M.; Sierra, J.M.; Miranda, M.; Peinado, F. Developing an automated planning tool for non-player character behavior. *CEUR Workshop Proc.* **2020**, *2719*, 69–77.
17. Baena-Perez, R.; Ruiz-Rube, I.; Dodero, J.M.; Bolivar, M.A. A Framework to Create Conversational Agents for the Development of Video Games by End-Users. In *Optimization and Learning*; Dorronsoro, B., Ruiz, P., de la Torre, J., Urda, D., Talbi, E.G., Eds.; Springer: Cham, Switzerland, 2020; p. 1173. [CrossRef]
18. Wu, Y.; Huo, Y.; Gao, Q.; Wu, Y.; Li, X. Game-theoretic and Learning-aided Physical Layer Security for Multiple Intelligent Eavesdroppers. In Proceedings of the 2022 IEEE GLOBECOM Workshops, Rio de Janeiro, Brazil, 4–8 December 2022; pp. 233–238.
19. Hu, M.; Weng, D.; Chen, F.; Wang, Y. Object Detecting Augmented Reality System. In Proceedings of the 2020 IEEE 20th International Conference on Communication Technology (ICCT), Nanning, China, 28–31 October 2020; pp. 1432–1438. [CrossRef]
20. Cerqueira, J.M.; Cleto, B.; Moura, J.M.; Sylla, C.; Ferreira, L. Potentiating Learning through Augmented Reality and Serious Games. In *Springer Handbook of Augmented Reality*; Springer: Berlin/Heidelberg, Germany, 2023; p. 369.
21. Elkoubaiti, H.; Mrabet, R. A Survey of Pedagogical Affordances of Augmented and Virtual Realities Technologies in loT–Based Classroom. In Proceedings of the IEEE 5th International Congress on Information Science and Technology (CiSt), Marrakech, Morocco, 21–27 October 2018; pp. 334–341.
22. Penty, C. Behind the scenes of The Cavern UE5 Cinematic Visual Tech Test. In Proceedings of the SIGGRAPH, Los Angeles, CA, USA, 8–11 August 2022; p. 4. [CrossRef]
23. Estrada, J.; Paheding, S.; Yang, X.; Niyaz, Q. Deep-Learning-Incorporated Augmented Reality Application for Engineering Lab Training. *Appl. Sci.* **2022**, *12*, 5159. [CrossRef]
24. Sang, F.; Wu, H.; Liu, Z.; Fang, S. Digital Twin Platform Design for Zhejiang Rural Cultural Tourism Based on Unreal Engine. In Proceedings of the 2022 International Conference on Culture-Oriented Science and Technology, CoST 2022, Lanzhou, China, 18–21 August 2022; pp. 274–278.
25. Liubogoshchev, M.; Ragimova, K.; Lyakhov, A.; Tang, S.; Khorov, E. Adaptive Cloud-Based Extended Reality: Modeling and Optimization. *IEEE Access* **2021**, *9*, 1. [CrossRef]
26. Zhang, J.; Li, H.; Teng, Y.; Zhang, R.; Chen, Q.; Chen, G. Research on the Application of Artificial Intelligence in Games. In Proceedings of the 9th International Conference on Digital Home, Guangzhou, China, 28–30 October 2022; pp. 207–212.
27. Paduraru, C.; Paduraru, M.; Stefanescu, A. RiverGame—A game testing tool using artificial intelligence. In Proceedings of the 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), Valencia, Spain, 4–14 April 2022; pp. 422–432. [CrossRef]
28. Vitek, M.; Peer, P. Intelligent agents in games: Review with an open-source tool. *Adv. Comput.* **2020**, *116*, 251–303. [CrossRef]
29. Liu, R.; Li, H.; Lv, Z. Modeling Methods of 3D Model in Digital Twins. *CMES Comput. Model. Eng. Sci.* **2023**, *136*, 985–1022.
30. Yao, R.; Zhang, W.; Liu, H. Design and Implementation of Real-Time 3D Interactive System Based on Unreal Engine. In *ACM International Conference Proceeding Series*; ACM: New York, NY, USA, 2022; pp. 30–36. [CrossRef]

31. Tavares, R.I.; Da Costa, F.; Luizelli, M.C.; Petrangeli, S.; Torres Vega, M.; Van der Hooft, J.; Wauters, T.; De Turck, F.; Gaspary, L.P. Dissecting the Performance of VR Video Streaming through the VR-EXP Experimentation Platform. *ACM Trans. Multimed. Comput. Commun. Appl.* **2019**, *15*, 111. [CrossRef]
32. Kuzbass Intersectoral Center for Labor Protection. VR-Simulators/3D-Simulators on the ProExpVR Platform. Available online: https://kuzbasscot.ru/virtualnye-trenazhery/ (accessed on 1 March 2023).
33. DT Consulting. Case: VR Technologies in the Training Center of the Mine. Available online: https://www.dtconsulting.ru/case-vr-mine (accessed on 1 March 2023).
34. Babkov, V.S. Development of the Virtual Mine Silent Complex Based on the Microsoft Kinect platform—A Presentation. Available online: http://www.myshared.ru/slide/491418/ (accessed on 1 March 2023).
35. Bashkov, E.A.; Babkov, D.C. The software system for building a virtual working environment at mining enterprises. South Federal University Bulletin. *Tech. Sci. SFU* **2012**, *5*, 211–215.
36. Krajčovič, M.; Gabajová, G.; Furmannová, B.; Vavrík, V.; Gašo, M.; Matys, M. A Case Study of Educational Games in Virtual Reality as a Teaching Method of Lean Management. *Electronics* **2021**, *10*, 838. [CrossRef]
37. Khan, S. Data Visualization to Explore the Countries Dataset for Pattern Creation. *Int. J. Online Biomed. Eng.* **2021**, *17*, 4–19. [CrossRef]
38. Khan, S. Artificial Intelligence Virtual Assistants (Chatbots) are Innovative Investigators. *Int. J. Comput. Sci. Netw. Secur.* **2020**, *20*, 93–98.
39. Son, L.H.; Chiclana, F.; Raghavendra, K.; Mittal, M.; Khari, M.; Chatterjee, J.M.; Baik, S.W. ARM-AMO: An Efficient Association Rule Mining Algorithm Based on Animal Migration Optimization. *Knowl. Based Syst.* **2018**, *154*, 68–80. [CrossRef]
40. Mahajan, S.; Abualigah, L.; Pandit, A.K.; Altalhi, M. Hybrid Aquila optimizer with arithmetic optimization algorithm for global optimization tasks. *Soft Comput.* **2022**, *26*, 4863–4881. [CrossRef]
41. Sharma, D.K.; Utsha, S.; Gupta, A.; Khari, K. Modified minimum spanning tree based vertical fragmentation, allocation and replication approach in distributed multimedia databases. *Multimed. Tools Appl.* **2022**, *81*, 37101–37118. [CrossRef]
42. Zhironkina, O.; Zhironkin, S. Technological and Intellectual Transition to Mining 4.0: A Review. *Energies* **2023**, *16*, 1427. [CrossRef]
43. Abu-Abed, F.; Pivovarov, K.; Zhironkin, V.; Zhironkin, S. Development of a Software Tool for Visualizing a Mine (Wellbore) in the Industrial Drilling of Oil Wells. *Processes* **2023**, *11*, 624. [CrossRef]
44. Abu-Abed, F. Development of Three-Dimensional Models of Mining Industry Objects. *E3S Web Conf.* **2021**, *278*, 01002.
45. Abu-Abed, F.; Khabarov, A. Development of pedestrian artificial intelligence utilizing unreal engine 4 graphic. *Int. J. Recent Technol. Eng.* **2019**, *8*, 639–642.