UDC 004.9; 004.94
DOI https://doi.org/10.32782/IT/2024-3-8

Vitaliia KOIBICHUK
PhD Economics, Associate Professor, Head of the Department of Economic Cybernetics, Sumy State University, 57, Petropavlivska Str., Sumy, Ukraine, 40000, v.koibichuk@biem.sumdu.edu.ua
***ORCID:** 0000-0002-3540-7922*

Roman KOCHEREZHCHENKO
Master's Student, Sumy State University, 57, Petropavlivska Str., Sumy, Ukraine, 40000, r.kocherezhchenko@student.sumdu.edu.ua
***ORCID:** 0000-0001-7269-4177*

Kostiantyn HRYTSENKO
Candidate of Technical Sciences, Associate Professor at the Department of Economic Cybernetics, Sumy State University,, 57, Petropavlivska Str., Sumy, Ukraine, 40000, k.hrytsenko@biem.sumdu.edu.ua
***ORCID:** 0000-0002-7855-691X*

Valerii YATSENKO
Candidate of Technical Sciences, Associate Professor at the Department of Economic Cybernetics, Sumy State University, 57, Petropavlivska Str., Sumy, Ukraine, 40000, v.yatsenko@biem.sumdu.edu.ua
***ORCID:** 0000-0003-2316-3817*

Alina YEFIMENKO
PhD, Assistant at the Department of Economic Cybernetics, Sumy State University, 57, Petropavlivska Str., Sumy, Ukraine, 40000, a.yefimenko@uabs.sumdu.edu.ua
***ORCID:** 0000-0002-2810-0965*

To cite this article: Koibichuk, V., Kocherezhchenko, R., Hrytsenko, K., Yatsenko, V., Yefimenko, A. (2024). Alhorytmy protsedurnoi heneratsii ihrovoho kontentu za dopomohoiu hrafiv [Algorithms for procedural generation of game content using graphs]. *Information Technology: Computer Science, Software Engineering and Cyber Security*, 3, 77–87, doi: https://doi.org/10.32782/IT/2024-3-8

ALGORITHMS FOR PROCEDURAL GENERATION OF GAME CONTENT USING GRAPHS

Developing unique gaming environments using algorithms based on graph data structures and procedural content generation can significantly reduce costs while increasing overall team productivity and eliminating the risk of stagnation in the development process.

***The purpose** of this research is to analyze, develop and visualize the operation of procedural content generation algorithms, as well as to study the prospects for their further use in the practical development of game projects.*

***The scientific novelty** is to use graphs for procedural generation of game content. This topic was chosen due to the fact that creating a game environment can be one of the main and most resource-intensive costs in the game production process. Procedural content generation can reduce these costs and speed up the development process. In fact, it is almost impossible to calculate what specific part of the team's productivity and business benefits procedural generation brings, since most discoveries in this area are a trade secret of most game studios, however, this only speaks of the opportunities and benefits that this approach brings.*

***The methodology** is based on The Python programming language as the main tool for studying algorithms, which was used to develop algorithms, create visualizations and examples of web servers for processing data generated by graphs.*

***Conclusion:** during development, differences and commonalities in the details of the implementation of algorithms, as well as the results of content generation, were studied. Differences in the generated graphs were also demonstrated. Examples of web servers illustrate the potential for further practical application of the developed algorithms. The results of the study can be used by developers of gaming environments and algorithms researchers to improve the efficiency of production processes.*

***Key words:** procedural generation, game development, development efficiency, development optimization.*

Віталія КОЙБІЧУК
кандидат економічних наук, доцент, завідувач кафедри економічної кібернетики, Сумський державний університет, вул. Петропавлівська, 57, м. Суми, Україна, 40000
***ORCID:** 0000-0002-3540-7922*

Роман КОЧЕРЕЖЧЕНКО
магістр, Сумський державний університет, вул. Петропавлівська, 57, м. Суми, Україна, 40000
***ORCID:** 0000-0001-7269-4177*

Костянтин ГРИЦЕНКО
кандидат технічних наук, доцент кафедри економічної кібернетики, Сумський державний університет, вул. Петропавлівська, 57, м. Суми, Україна, 40000
***ORCID:** 0000-0002-7855-691X*

Валерій ЯЦЕНКО
кандидат технічних наук, доцент кафедри економічної кібернетики, Сумський державний університет, вул. Петропавлівська, 57, м. Суми, Україна, 40000
***ORCID:** 0000-0003-2316-3817*

Аліна ЄФІМЕНКО
доктор філософії, асистент кафедри економічної кібернетики, Сумський державний університет, вул. Петропавлівська, 57, м. Суми, Україна, 40000
***ORCID:** 0000-0002-2810-0965*

Бібліографічний опис статті: Койбічук, В., Кочережченко, Р., Гриценко, К., Яценко, В., Єфіменко, А. (2024). Алгоритми процедурної генерації ігрового контенту за допомогою графів. *Information Technology: Computer Science, Software Engineering and Cyber Security*, 3, 77–87, doi: https://doi.org/10.32782/IT/2024-3-8

АЛГОРИТМИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ ІГРОВОГО КОНТЕНТУ З ВИКОРИСТАННЯМ ГРАФІВ

Розробка унікальних ігрових середовищ з використанням алгоритмів на основі графових структур даних та процедурної генерації контенту дозволяє суттєво скоротити витрати при одночасному підвищенні загальної продуктивності команди та усуненні ризику стагнації процесу розробки.

***Метою роботи** є аналіз, розробка та візуалізація роботи алгоритмів процедурної генерації контенту, а також вивчення перспектив їх подальшого використання у практичній розробці ігрових проектів.*

***Наукова новизна** полягає у використанні графів для процедурної генерації ігрового контенту. Ця тема була обрана у зв'язку з тим, що створення ігрового оточення може бути однією з основних і найбільш ресурсоємних витрат у процесі виробництва гри. Процедурна генерація контенту може зменшити ці витрати та пришвидшити процес розробки. Практично неможливо підрахувати, яку конкретно частину продуктивності команди та бізнес-вигоди приносить процедурна генерація, оскільки більшість інновацій у цій сфері є комерційною таємницею ігрових студій, однак це говорить лише про можливості та переваги, які несе в собі цей підхід.*

***Методологія** базується на мові програмування The Python як основному інструменті для вивчення алгоритмів, який використовувався для розробки алгоритмів, створення візуалізацій та прикладів веб-серверів для обробки даних, згенерованих графами.*

***Висновок:** під час розробки були вивчені відмінності та спільні риси в деталях реалізації алгоритмів, а також результати генерації контенту. Також було продемонстровано відмінності у згенерованих графах. На прикладах веб-серверів проілюстровано потенціал подальшого практичного застосування розроблених алгоритмів. Результати дослідження можуть бути використані розробниками ігрових середовищ та дослідниками алгоритмів для підвищення ефективності виробничих процесів.*

***Ключові слова:** процедурна генерація, розробка ігор, ефективність розробки, оптимізація розробки.*

Introduction. Procedural content generation (PGC) is considered one of the tools for creating a unique player experience in projects of various scales. Minimizing the cost of game design development, it allows you to automate the generation of game maps, dialogues and events (Togelius et al., 2011). Thus, the content of the game world, the environment of the character is enriched and opportunities for dynamic adaptation to the user's actions are created. The effective application of

PCC can compensate for shortcomings in design, gameplay mechanics, etc., and make such a gaming experience a special feature of the product (Van Der Linden et al., 2013).

Previously, the limited computing power of systems forced developers to save on memory and processor time, which prevented the use of this tool. However, with the development of technology, procedural generation has become not only possible, but also desirable for creating large and varied game universes.

In the context of traditional game design, where game environments are hand-crafted by teams of specialists, PGK offers an alternative approach that allows for the generation of infinitely diverse game scenarios and worlds. Using this approach not only reduces the dependence on the creative resources of the team and the efficiency of manual labor, but also opens new horizons for innovation in the industry. As a result, game implementation becomes more accessible to developers of various levels, while players gain access to inexhaustible content.

This study attracts special attention because it provides a unique opportunity to explore complex algorithmic and mathematical principles in the context of their practical application for creating dynamic and exciting game content. Using graphs as a basis for procedural generation allows modeling complex structures and relationships, opening new horizons for automating the creation of game worlds, levels, storylines, and dynamic game events. In addition, the PGK direction promotes the development of new methods of optimization and data analysis, as well as the application of graph theory in non-standard fields, such as data visualization and machine learning. This makes the researched technology very promising from both an academic and a practical point of view. In light of the constant development of the industry, its numerous innovations and the improvement of the quality of the gaming experience, the effective application of such algorithms can significantly expand the boundaries of what is possible in the development of game design

Statement of the problem. During the review of the literature, several works were studied to research issues related to procedural generation, its features, characteristics and opportunities for process optimization. To study procedural generation in general, its goals, capabilities and features, the following works were considered. In fact, there are not many scientific papers from which you can get the latest and most relevant information, since most technical open-source work takes place in game studios where most do not share the source code for analyzing algorithms. The first work that gives a general understanding of algorithms is «Procedural content generation for games: A survey», this material gave a general idea of the features of algorithms and their possible impact on development processes (Hendrikx et al., 2013). Another work, «Procedural content generation: Goals, challenges and actionable steps» allowed us to delve deeper not only into the technical details and challenges that a team that decides to use generative approaches in development may encounter, but also without the value that, with the right approach, gives a tangible increase in the productivity of the development team (Togelius et al., 2013). «What is procedural content generation? Mario on the borderline» gave a more thorough technical overview of the use of procedural generation in conditions of limited resources, which also gave insight into the possibilities of generating game landscapes of relatively large sizes, which is not possible in standard game design (Togelius et al., 2011). The remaining works mentioned in the work served as the technical and theoretical basis for the development and demonstration of algorithms. With their help, a strong and reliable basis was obtained for the practical implementation of algorithms using the graph data structure. **Thereby, the purpose of this research** is to analyze, develop and visualize the operation of procedural content generation algorithms, as well as to study the prospects for their further use in the practical development of game projects.

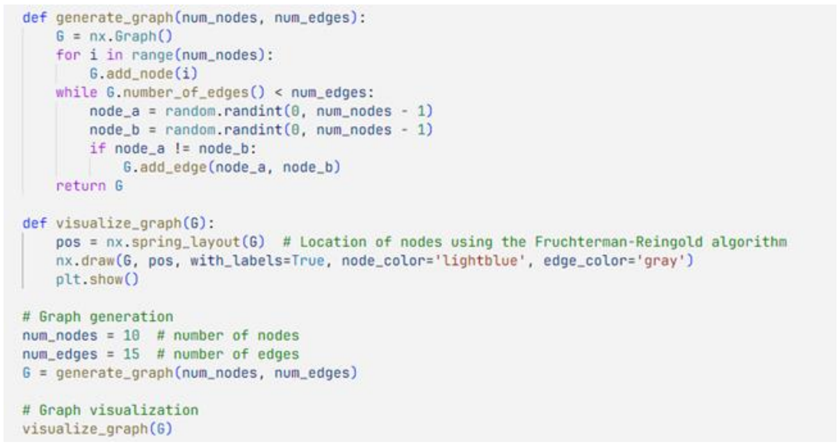
Methodology. For practical implementation, the general purpose programming language Python is used. It is chosen as one of the most popular languages, which has a convenient infrastructure with a large number of useful packages that simplify the solution of tasks. However, any application programming language can be used to implement these algorithms.

The development of all algorithms within this work will take place in several stages:

- Demonstration of pseudocode.
- Demonstration of working code in the chosen programming language.
- Visualization of the algorithm.

This will give understanding which algorithm is better to use for the needs of game designers.

To begin with, before the implementation of more complex algorithms, it is possible to show an easy version of the work of the library algorithm to demonstrate the practical possibilities of generation and visualization. The program code is shown in Figure 1, the results of the work on Figure 2. Having a basic understanding of the operation of some algorithms, as well as a visual demonstration



```python
def generate_graph(num_nodes, num_edges):
    G = nx.Graph()
    for i in range(num_nodes):
        G.add_node(i)
    while G.number_of_edges() < num_edges:
        node_a = random.randint(0, num_nodes - 1)
        node_b = random.randint(0, num_nodes - 1)
        if node_a != node_b:
            G.add_edge(node_a, node_b)
    return G

def visualize_graph(G):
    pos = nx.spring_layout(G)  # Location of nodes using the Fruchterman-Reingold algorithm
    nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray')
    plt.show()

# Graph generation
num_nodes = 10  # number of nodes
num_edges = 15  # number of edges
G = generate_graph(num_nodes, num_edges)

# Graph visualization
visualize_graph(G)
```

Fig. 1. Program code for implementing the library algorithm for graph generation and visualization

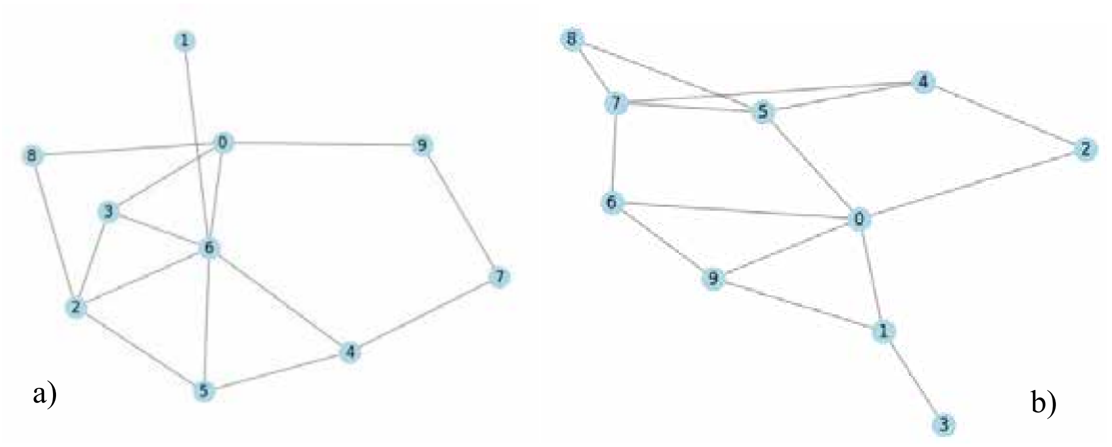


Fig. 2. The first result (a) and second result (b) of the work of the library algorithm

of the generated graph, that can help implementing algorithms.

Results. In order to generate a pseudocode (Figure 3) of the algorithm, we will generate the main steps of its operation:

- Graph initialization: We start with an empty graph and an initial node located at the origin of coordinates (0,0).
- Setting Directions: Define possible movements in the graph, which include movement to the left, right, down and up.
- Cycle for Random Wandering:
  - o Direction and Weight: At each iteration, we randomly select one of the specified directions and generate a random weight that affects the distance of the next step.
  - o Calculation of the Position of the Next Node: We determine the position of the next node using the position of the current node, to which we add the selected direction multiplied by the weight.

When the main steps are formed, we proceed to the formation of the pseudocode for the implementation of the algorithm.

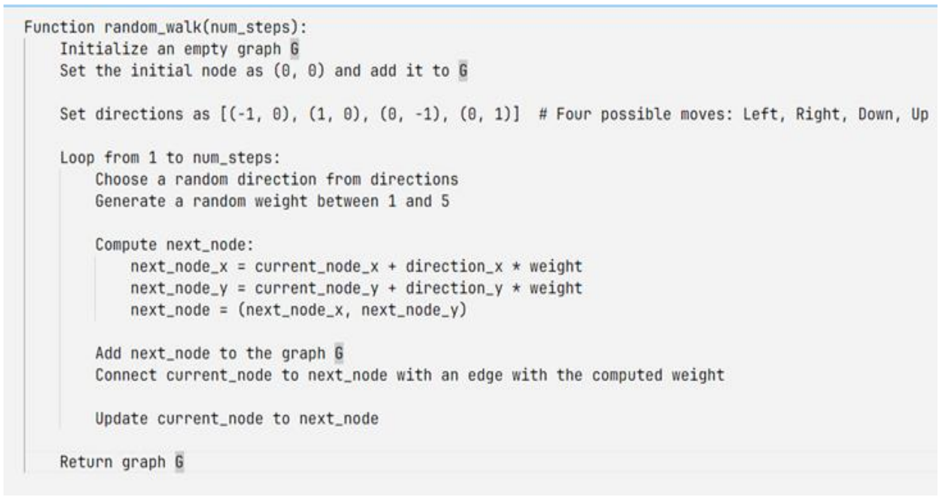
This is one of the simplest variants of the algorithm, while the algorithm itself is quite simple. However, it should be borne in mind that without additional configuration, the result will be quite simple for the structure of the game environment.

Next step is the implementation the algorithm using the selected programming language Figure 4.

We will test the described algorithm and create the generation results. For clarity, we will perform two generation of Figure 5. According to the results shown in the corresponding figures, it can be concluded that the algorithm is suitable for non-deterministic game environments. That is, it can be used as a basis for other algorithms, or as an additional step in the graph processing chain.

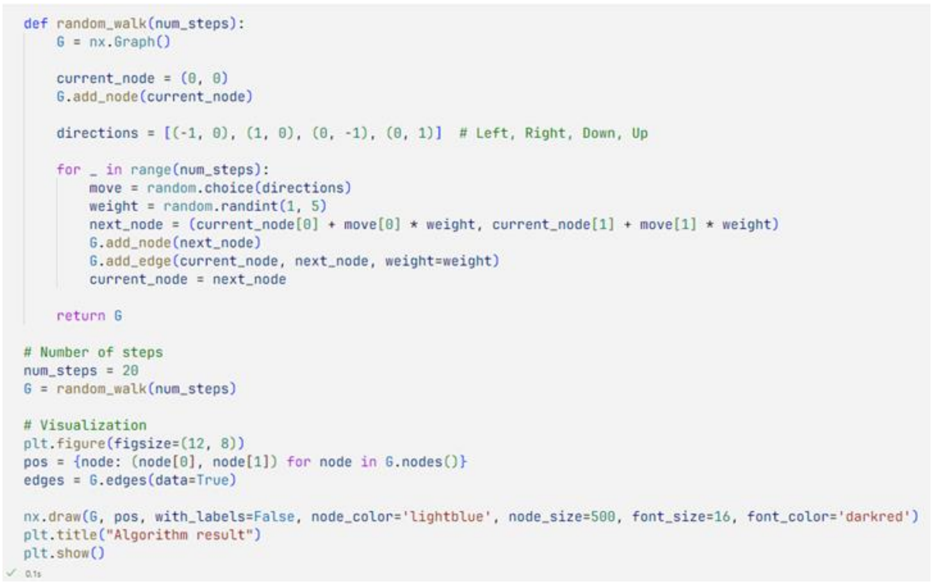
Now we practically implement the algorithm of binary division of space. First, let's form the main steps of the algorithm:

- Initialization: the initial area to be divided is selected.
- Recursive division: the selected area is divided into two parts using a straight line (or plane



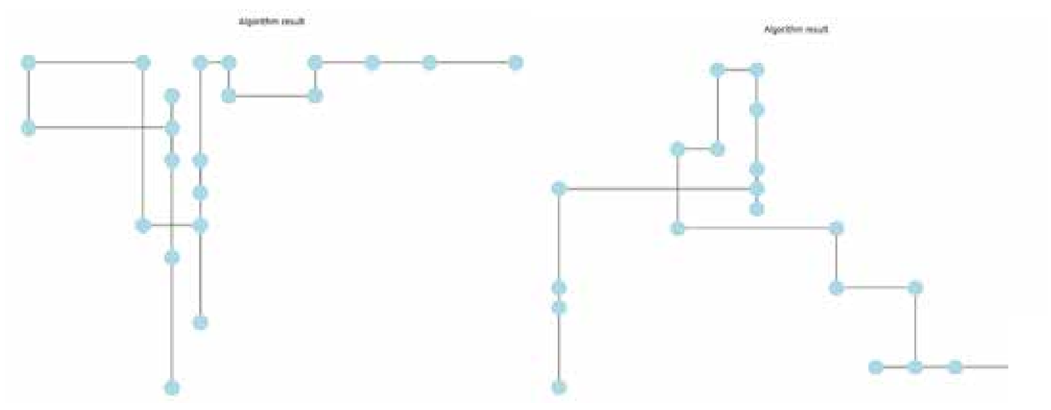
```
Function random_walk(num_steps):
    Initialize an empty graph G
    Set the initial node as (0, 0) and add it to G

    Set directions as [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Four possible moves: Left, Right, Down, Up

    Loop from 1 to num_steps:
        Choose a random direction from directions
        Generate a random weight between 1 and 5

        Compute next_node:
            next_node_x = current_node_x + direction_x * weight
            next_node_y = current_node_y + direction_y * weight
            next_node = (next_node_x, next_node_y)

        Add next_node to the graph G
        Connect current_node to next_node with an edge with the computed weight

        Update current_node to next_node

    Return graph G
```

Fig. 3. Drunkard's Walk algorithm pseudocode



```python
def random_walk(num_steps):
    G = nx.Graph()

    current_node = (0, 0)
    G.add_node(current_node)

    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Left, Right, Down, Up

    for _ in range(num_steps):
        move = random.choice(directions)
        weight = random.randint(1, 5)
        next_node = (current_node[0] + move[0] * weight, current_node[1] + move[1] * weight)
        G.add_node(next_node)
        G.add_edge(current_node, next_node, weight=weight)
        current_node = next_node

    return G

# Number of steps
num_steps = 20
G = random_walk(num_steps)

# Visualization
plt.figure(figsize=(12, 8))
pos = {node: (node[0], node[1]) for node in G.nodes()}
edges = G.edges(data=True)

nx.draw(G, pos, with_labels=False, node_color='lightblue', node_size=500, font_size=16, font_color='darkred')
plt.title("Algorithm result")
plt.show()
```

Fig. 4. Implementation of the algorithm in Python



a) b)

Fig. 5. The first result (a) and second result (b) of the Drunkard's Walk algorithm

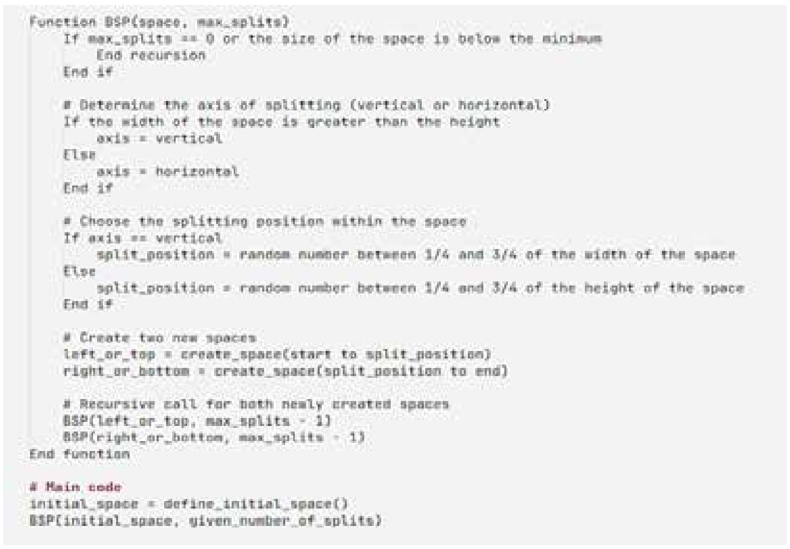
in 3D), which can pass vertically, horizontally or at any angle, depending on the task and the selection algorithm. This process continues recursively for each of the newly created parts until given criteria such as minimum part size are met.

- Stopping the algorithm: the recursion stops when each part reaches a certain minimum size or when the number of recursive divisions reaches a maximum limit.

We will describe the formed steps with the help of pseudocode Figure 6.

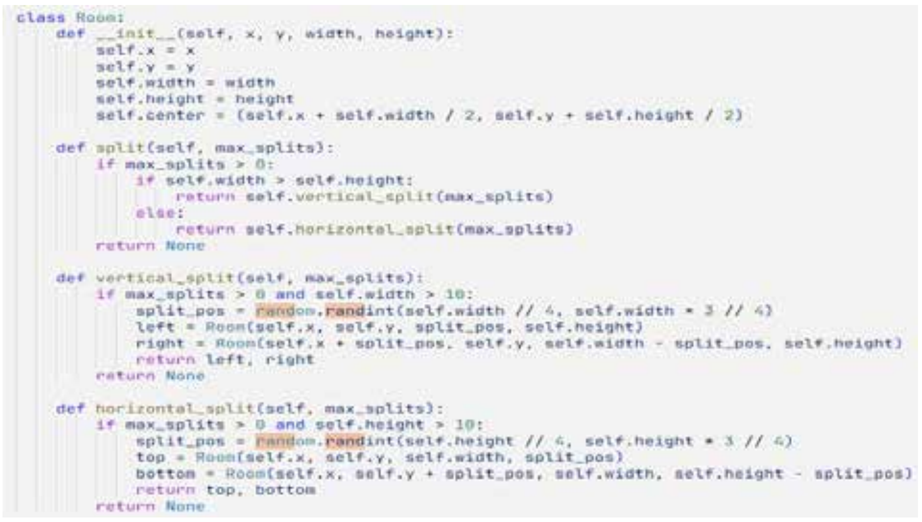
The main steps have been formed, let's move on to the software implementation using the selected programming language. Since the implementation turns out to be very voluminous – we will take it to the applications. Here we will describe exactly how the developed software part works:

- Room class – this class represents a room with given coordinates (x, y), width and height. It also calculates the center of the room, which is used for further calculations.
- The «_init_» method initializes the room with the given parameters and determines its center (Fig. 7).
- The «split» method divides a room into two smaller rooms (vertically or horizontally) based on which is greater: width or height. The choice of partitioning method depends on the aspect ratio of the room and the number of allowed partitions (max_splits) (Fig. 7).
- The «vertical_split» and «horizontal_split» methods perform their own splitting. They choose a position to break (not too close to the edges), create new rooms and return them (Fig. 7).
- «create_rooms» function – this function recursively divides the initial room into smaller ones until the maximum number of rooms is reached or until the possibility of division is exhausted. This



```
Function BSP(space, max_splits)
    If max_splits == 0 or the size of the space is below the minimum
        End recursion
    End if

    # Determine the axis of splitting (vertical or horizontal)
    If the width of the space is greater than the height
        axis = vertical
    Else
        axis = horizontal
    End if

    # Choose the splitting position within the space
    If axis == vertical
        split_position = random number between 1/4 and 3/4 of the width of the space
    Else
        split_position = random number between 1/4 and 3/4 of the height of the space
    End if

    # Create two new spaces
    left_or_top = create_space(start to split_position)
    right_or_bottom = create_space(split_position to end)

    # Recursive call for both newly created spaces
    BSP(left_or_top, max_splits - 1)
    BSP(right_or_bottom, max_splits - 1)
End function

# Main code
initial_space = define_initial_space()
BSP(initial_space, given_number_of_splits)
```

Fig. 6. Pseudocode for implementing the Binary Space Partitioning algorithm



```python
class Room:
    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.center = (self.x + self.width / 2, self.y + self.height / 2)

    def split(self, max_splits):
        if max_splits > 0:
            if self.width > self.height:
                return self.vertical_split(max_splits)
            else:
                return self.horizontal_split(max_splits)
        return None

    def vertical_split(self, max_splits):
        if max_splits > 0 and self.width > 10:
            split_pos = random.randint(self.width // 4, self.width * 3 // 4)
            left = Room(self.x, self.y, split_pos, self.height)
            right = Room(self.x + split_pos, self.y, self.width - split_pos, self.height)
            return left, right
        return None

    def horizontal_split(self, max_splits):
        if max_splits > 0 and self.height > 10:
            split_pos = random.randint(self.height // 4, self.height * 3 // 4)
            top = Room(self.x, self.y, self.width, split_pos)
            bottom = Room(self.x, self.y + split_pos, self.width, self.height - split_pos)
            return top, bottom
        return None
```

Fig. 7. Functions from BSP implementation: --init--, split, vertical_split, horizontal_split

provides control over the number of rooms in the final structure (Fig. 8).

• «build_graph» function – this function creates a graph where each room is a node. Nodes are connected by edges based on the distance between room centers, but only until the number of connections per room exceeds the specified limit (max_connections_per_room). This limits the degree of connectivity between rooms, which can be useful for creating more realistic room layouts (Fig. 9).

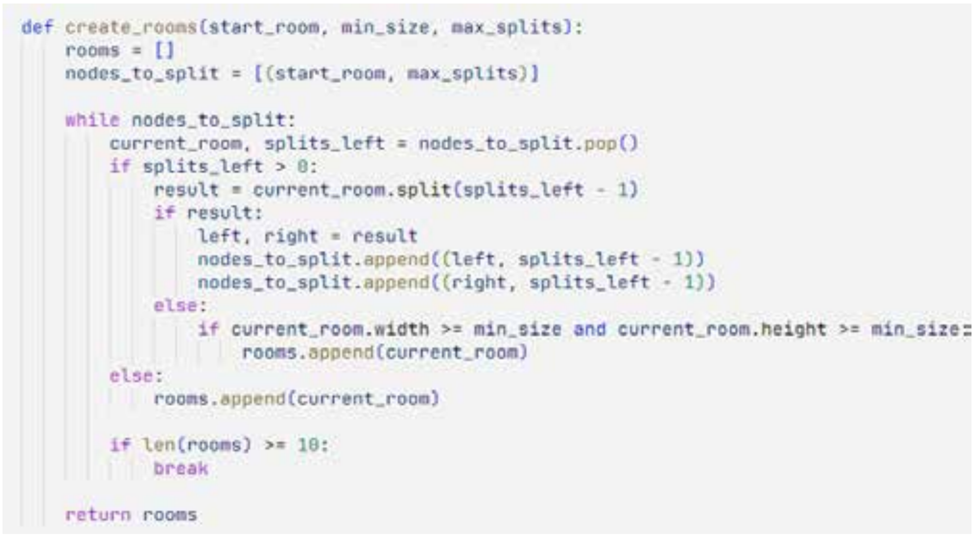
• «draw_graph» function – this function renders a graph using the NetworkX graph drawing library. It uses room positions to place nodes and shows the connections between them (Fig. 10).

Figure 11 shows running and visualization of the algorithm.

General execution flow: an initial room is created, it is divided into smaller rooms using the «create_rooms» function, After the division of rooms is complete, a graph is created from these rooms, the graph is visualized using the «draw_graph» function. The results of the algorithm can be seen in Figure 12.

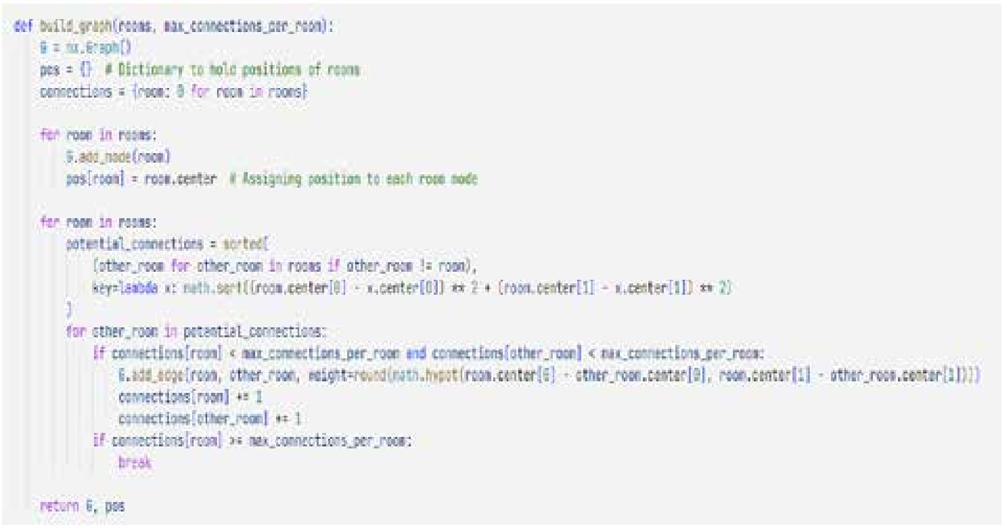
From the obtained results, we can see that this particular algorithm is more suitable for practical problems, because it has the ability to fine-tune parameters that allow you to obtain different, but predictable results.

The practical application of graph generation algorithms consists in the use of generated structures – as a basic representation of game environments. However, simple visualization of the generated graph is of no practical value for further use. To obtain usable results, it is advisable to implement a program interface (API – Application Programming Interface) that provides output



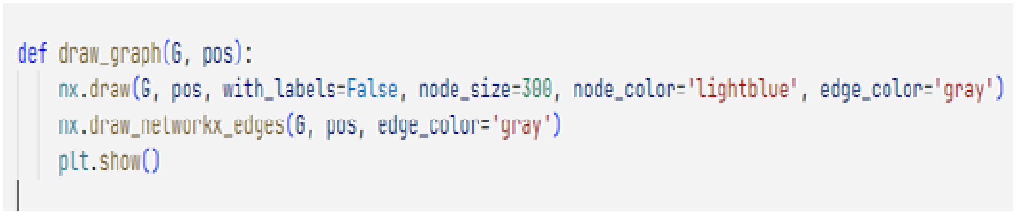
```
def create_rooms(start_room, min_size, max_splits):
    rooms = []
    nodes_to_split = [(start_room, max_splits)]

    while nodes_to_split:
        current_room, splits_left = nodes_to_split.pop()
        if splits_left > 0:
            result = current_room.split(splits_left - 1)
            if result:
                left, right = result
                nodes_to_split.append((left, splits_left - 1))
                nodes_to_split.append((right, splits_left - 1))
            else:
                if current_room.width >= min_size and current_room.height >= min_size:
                    rooms.append(current_room)
        else:
            rooms.append(current_room)

        if len(rooms) >= 10:
            break

    return rooms
```

Fig. 8. BSP implementation functions: create_rooms



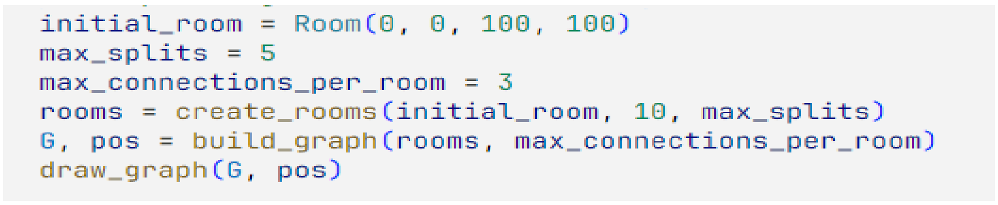
```
def build_graph(rooms, max_connections_per_room):
    G = nx.Graph()
    pos = {}  # Dictionary to hold positions of rooms
    connections = {room: 0 for room in rooms}

    for room in rooms:
        G.add_node(room)
        pos[room] = room.center  # Assigning position to each room node

    for room in rooms:
        potential_connections = sorted(
            [other_room for other_room in rooms if other_room != room],
            key=lambda x: math.sqrt((room.center[0] - x.center[0]) ** 2 + (room.center[1] - x.center[1]) ** 2)
        )
        for other_room in potential_connections:
            if connections[room] < max_connections_per_room and connections[other_room] < max_connections_per_room:
                G.add_edge(room, other_room, weight=round(math.hypot(room.center[0] - other_room.center[0], room.center[1] - other_room.center[1])))
                connections[room] += 1
                connections[other_room] += 1
            if connections[room] >= max_connections_per_room:
                break

    return G, pos
```

Fig. 9. Functions from BSP implementation: build_graph



```
def draw_graph(G, pos):
    nx.draw(G, pos, with_labels=False, node_size=300, node_color='lightblue', edge_color='gray')
    nx.draw_networkx_edges(G, pos, edge_color='gray')
    plt.show()
```

Fig. 10. The draw_graph function



```
initial_room = Room(0, 0, 100, 100)
max_splits = 5
max_connections_per_room = 3
rooms = create_rooms(initial_room, 10, max_splits)
G, pos = build_graph(rooms, max_connections_per_room)
draw_graph(G, pos)
```

Fig. 11. Running and visualization of the algorithm

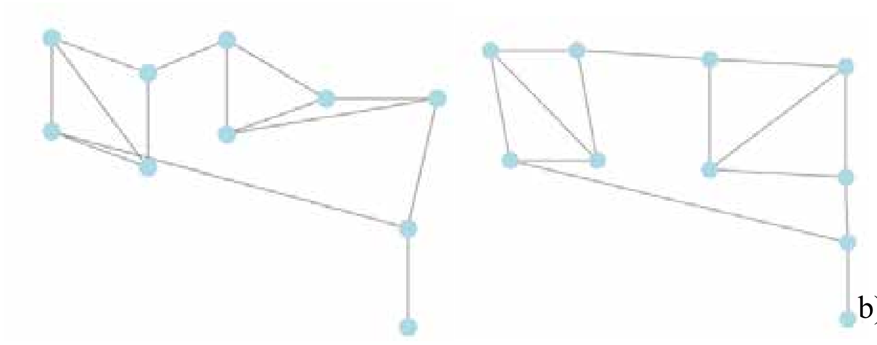


Fig. 12. The first result (a) and second result (b) of the BSP algorithm

data of the algorithm in a standardized format – for example, JSON.

API defines a set of interaction rules between software components, ensuring their compatibility and efficient data exchange. To create it, without complicating the basic code of the project, you can use the Flask web framework. It is designed for building simple web applications using the Python language. This tool is minimalistic, modular and scalable. The structure of the Flask program includes routing of URL requests, processing of HTTP methods (GET, POST, PUT, DELETE). It allows you to efficiently create RESTful APIs with CRUD functionality (create, read, update, delete data).

To begin with, let's create a simple server with the GET method to retrieve data from the graph. The implementation of the web server is shown in Figure 13. Here we import the necessary modules and then create a Flask instance. We leave the Generate_graph function unchanged.

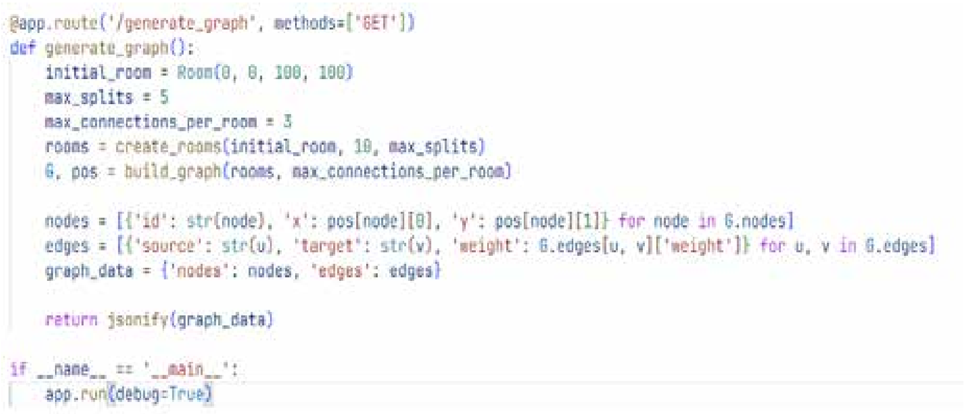
Next, we define the route /graph/<int:num_nodes>/<int:num_edges> with the HTTP GET method. In this function, we will generate a graph using generate_graph. Then we convert it into node-link format using nx.node_link_data and return the result in JSON format using jsonify.

To run the program, execute the command python simple_graph.py in the terminal. As a trace, we get the JSON object of the graph by sending a GET request to «http://localhost:5000/graph/<num_nodes>/<num_edges>», where <num_nodes> and <num_edges> are replaced with the desired numbers.

For example, a query for «http://localhost:5000/graph/3/3» will return a graph JSON object with 3 nodes and 3 edges. The result of executing such a request is shown in Figure 14.

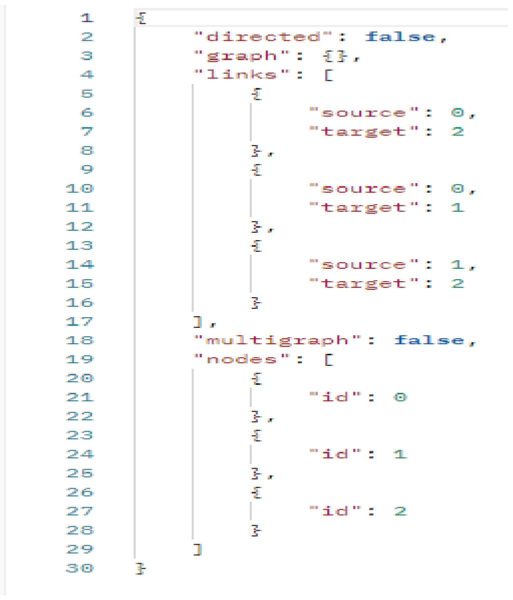
In this implementation, the placement of objects is not detailed, which opens perspectives for the development of a web server adapted to the specific requirements of users.

Similar to this implementation of the algorithm, other algorithms can be created to obtain similar graph structures. The next example of the implementation of the API for receiving data will create a server with a GET request to receive the results of generation according to the BSP (Binary Space Partitioning) algorithm. The implementation is described Figure 15.



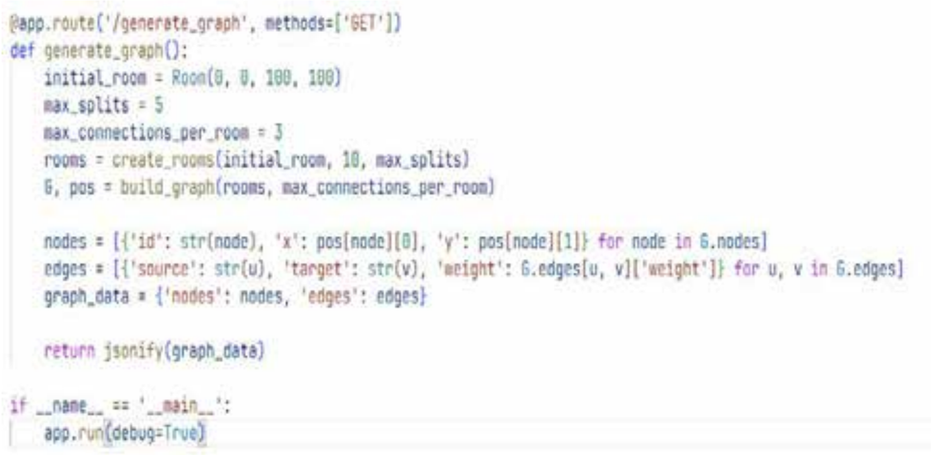
```python
@app.route('/generate_graph', methods=['GET'])
def generate_graph():
    initial_room = Room(0, 0, 100, 100)
    max_splits = 5
    max_connections_per_room = 3
    rooms = create_rooms(initial_room, 10, max_splits)
    G, pos = build_graph(rooms, max_connections_per_room)

    nodes = [{'id': str(node), 'x': pos[node][0], 'y': pos[node][1]} for node in G.nodes]
    edges = [{'source': str(u), 'target': str(v), 'weight': G.edges[u, v]['weight']} for u, v in G.edges]
    graph_data = {'nodes': nodes, 'edges': edges}

    return jsonify(graph_data)

if __name__ == '__main__':
    app.run(debug=True)
```

Fig. 13. Web server route to receive BSP generation results



```json
{
    "directed": false,
    "graph": {},
    "links": [
        {
            "source": 0,
            "target": 2
        },
        {
            "source": 0,
            "target": 1
        },
        {
            "source": 1,
            "target": 2
        }
    ],
    "multigraph": false,
    "nodes": [
        {
            "id": 0
        },
        {
            "id": 1
        },
        {
            "id": 2
        }
    ]
}
```

Fig. 14. The result of the request to receive graph data



```python
@app.route('/generate_graph', methods=['GET'])
def generate_graph():
    initial_room = Room(0, 0, 100, 100)
    max_splits = 5
    max_connections_per_room = 3
    rooms = create_rooms(initial_room, 10, max_splits)
    G, pos = build_graph(rooms, max_connections_per_room)

    nodes = [{'id': str(node), 'x': pos[node][0], 'y': pos[node][1]} for node in G.nodes]
    edges = [{'source': str(u), 'target': str(v), 'weight': G.edges[u, v]['weight']} for u, v in G.edges]
    graph_data = {'nodes': nodes, 'edges': edges}

    return jsonify(graph_data)

if __name__ == '__main__':
    app.run(debug=True)
```

Fig. 15. Web server route to receive BSP generation results

In this example, we create a Flask server with a single route /generate_graph. When executing a request with this route, the server generates a graph using the create_rooms and build_graph functions. It then converts the graph to JSON format using list inclusions. Graph nodes are represented as dictionaries with id, x, and y keys, and edges are represented as dictionaries with source, target, and weight keys. The result of the generation is the edge and nodes arrays. «Edge» array has objects of type { «source»: string, target: string, weight: number}. The «Node» array has objects of type {«id»: string, x: float, y: float}. We return the received graph data in JSON format using the jsonify function from Flask.

Conclusions. In conclusion, among the main vectors for further development of the project, optimization for more complex and branched graph structures for the purpose of detailed modeling of the game world should be highlighted. Achieving this goal will require improving algorithmic solutions that provide greater flexibility in settings and optimization for working with large and complex graph data structures. In addition, the issue of integration with game engines is critical, which will optimize the development process for the specific context of game engines and speed up overall development cycles.

To successfully complete these tasks, it is necessary to consider various optimization methods, such as the use of more efficient search and data processing algorithms, as well as the use of modern technologies that reduce computational costs. Integration with game engines requires close collaboration with the engine development teams and a deep understanding of their architecture and capabilities. It is also important to pay attention to user experience, developing intuitive interfaces and tools, improving documentation and providing training materials.

The support of the developer community plays a key role in the successful development of the project. An active and engaged community can significantly speed up the development process by facilitating the sharing of experiences, suggestions for improvements, and collaborative problem solving. Regular meetings, webinars and conferences dedicated to discussing the current status of the project and plans for the future will facilitate this process.

Thus, the successful development of the project requires an integrated approach, including optimization of graph structures, integration with game engines, improvement of user experience and active support of the developer community. This is the only way to achieve your goals and create an innovative product that can change the approach to game development and modeling of game worlds.

BIBLIOGRAPHY:

1. Xia F., Liu J., Nie H., Fu Y., Wan L. and Kong X. «Random Walks: A Review of Algorithms and Applications» in IEEE Transactions on Emerging Topics in Computational Intelligence, April 2020, Volume 4, No. 2, pp. 95–107, doi: 10.1109/TETCI.2019.2952908.

2. Fan X., Li B., Sisson S. The binary space partitioning-tree process. International Conference on Artificial Intelligence and Statistics, March 2018, PMLR, pp. 1859–1867.

3. Ehrhardt G. The not-so-random Drunkard's walk, Journal of Statistics Education, 2013, vol. 21, no. 2, doi: 10.1080/10691898.2013.11889679.

4. Hendrikx M., Meijer S., Van Der Velden, J., Iosup A. Procedural content generation for games: a survey, ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 2013, vol. 9, no. 1, p. 1–22.

5. Koesnaedi A., Istiono W. Implementation drunkard's walk algorithm to generate random level in roguelike games. International Journal of Multidisciplinary Research and Publications, 2022, vol. *5*, no. 2, p. 97–103.

6. Shaker N., Togelius J., Nelson M. J. Procedural content generation in games, 2016.

7. Togelius J., Kastbjerg E., Schedl D., Yannakakis G. N. What is procedural content generation? Mario on the borderline. Proceedings of the 2nd international workshop on procedural content generation in games, June 2011, pp. 1–6.

8. Cóth C. D. Binary space partitions: recent developments. Combinatorial and Computational Geometry, 2005, vol. 52, p. 525–552.

9. Van Der Linden R., Lopes R., Bidarra R. Procedural generation of dungeons. IEEE Transactions on Computational Intelligence and AI in Game*s*, 2013, vol. *6,* no. 1, p. 78–89.

REFERENCES:

1. Xia, F., Liu, J., Nie, H., Fu, Y., Wan, L. and Kong, X. (2020). «Random Walks: A Review of Algorithms and Applications» in IEEE Transactions on Emerging Topics in Computational Intelligence, April 2020, Volume 4, No. 2, pp. 95–107, doi: 10.1109/TETCI.2019.2952908 [in English].

2. Fan, X., Li, B., & Sisson, S. (2018). The binary space partitioning-tree process. International Conference on Artificial Intelligence and Statistics, March 2018, PMLR, pp. 1859–1867 [in English].

3. Ehrhardt, G. (2013). The not-so-random Drunkard’s walk, Journal of Statistics Education, vol. 21, no. 2, doi: 10.1080/10691898.2013.11889679 [in English].

4. Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural content generation for games: a survey, ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), vol. 9, no. 1, p. 1–22 [in English].

5. Koesnaedi, A., & Istiono, W. (2022). Implementation drunkard’s walk algorithm to generate random level in roguelike games. International Journal of Multidisciplinary Research and Publications, vol. *5*, no. 2, p. 97–103. [in English].

6. Shaker, N., Togelius, J., & Nelson, M. J. (2016). Procedural content generation in games [in English].

7. Togelius, J., Kastbjerg, E., Schedl, D., & Yannakakis, G. N. (2011). What is procedural content generation? Mario on the borderline. Proceedings of the 2nd international workshop on procedural content generation in games, June 2011, pp. 1–6 [in English].

8. Cóth, C. D. (2005). Binary space partitions: recent developments. Combinatorial and Computational Geometry, vol. 52, p. 525–552 [in English].

9. Van Der Linden, R., Lopes, R., & Bidarra, R. (2013). Procedural generation of dungeons. *IEEE* Transactions on Computational Intelligence and AI in Games, vol. *6,* no. 1, p. 78–89 [in English].