ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785
(ISRA), Journal Impact Factor: 2.114

IJESRT

INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

OBJECT ORIENTED GAME LEVEL GENERATOR

Rushil Agrawal*, Soumyadeep Ghosh
Manipal Institute of Technology, Manipal University, Manipal-576104, Karnataka, India

ABSTRACT

Level Generation deals with designing the contents of a game. Object Oriented approach and content generation together combined can be used to create game level designs. This paper discusses how to combine the two, using the famous game Mario Bros as and example. The paper presents work towards an object oriented universal level generator.

KEYWORDS: Content Recognition, Game Engine, Pattern Creation, Level Generator, Difficulty Management, OOP.

INTRODUCTION

This paper discusses the relation between Object Oriented content generation (OOCG) and design patterns in games, and presents a work on a universal level generator designed using Object Oriented concepts. Object oriented content generation in digital games refers to the automated or semi-automated creation of game content. Design patterns are a way of structuring the basic components and the design process into recurring elements. This way of thinking originated in architecture, has had major impact on software development and has recently been applied to game design. In recent days, with the birth of games like Temple Run, Subway Surfer, Flappy Birds etc., the game industry has seen a sudden rise of interest in infinite level games, i.e. games which have a probability of running till infinity. In the following paragraphs, we will discuss Object Oriented Content Generation and in particular its role in providing variation in game followed by discussion on design patterns, and how they can be used for OOCG. We then describe the procedure of building an automated infinite level generator.
We have also used an Object oriented language(Java) to write and explain sample codes.

OBJECT ORIENTED CONTENT GENERATION

Object Oriented Content Generation (OOCG) is the process of designing various entities of a game as different class. Objects generated from each class represent an instance of a component in the game. These objects have the capability to interact with each other. The result of interaction is to be decided by the game designer. For example, a class can be designed for the main character in the game, it has properties of life health, bonus, points collected etc.. Now, an object is instantiated each time the character is born and the object gets destroyed each time it dies. This also helps in data management and memory savings due to re-usability property of a class. An interesting example of data compression is the space trading game Elite where David Braben and Ian Bell succeeded to squeeze in 8 galaxies with 256 stars each into the limited (22 kB) memory capacity of the BBC Microcomputer with the help of pseudo-random numbers.

BASIC CONTENTS

Games are in many aspects a combination of designed structures. Rules govern the game's use in the play processing a way that creates meaning to players by allowing or disallowing actions. Levels guide the player through the game world, and sometimes, game space as well, because of its way of structuring challenges and rewards in the game space. The game's story builds meaning to the actions of non-player characters (NPCs) and other entities populating the game world and together with quests, the story provide purpose to the challenges the player is presented with. Thus we conclude that structures in games are fundamentally entities that are inter-connected with each other by relationships in a stable and consistent matter for a single game. They should also be observable and recognisable from the user's perspective, either for the player character (PC), on the player's experience level or beyond the game, like game genres and themes. It is from the different structures we define meaning and consistency in abstract things as digital games. Structure s operates much like context, and participates in the meaning-making process. By ordering the elements of

ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785
(ISRA), Journal Impact Factor: 2.114

a system in very particular ways, structure works to create meaning." Typically the designed structure is a functional one if helps to create meaning for the player. By analysing the structures in games we can and the reason to why a certain game makes meaning in a certain way. Digital games are relaying on its ability to convey information on the game state through visual output. In action games, more often than not, the player needs to scan, interpret and assess the information of the game state at high speed. The assessment of the game state allows the player to interact with the game. In design practice the user's ability to act is conveyed through affordances and the user's inability to act is conveyed through constraints. In order to be able to generate suitable content for a digital game the algorithm has to be able to produce output that is meaningful to the player. Unless the player is able to understand the output's meaning it will be experienced as noise or make the player chose a less advantageous or perhaps even make the player fail. In the process of game development the meaning and structure is expressed in the content the game designer and level designer provides. The game designer provides the development team with the overall picture of how the game should function. Level designers use a toolkit or "level editor" to develop new missions, scenarios, or quests for the players. They lay out the components that appear on the level or map and work closely with the game designer to make these into the overall theme of the game. However, this job is not an easy one because for some Level design is an art. We believe that a helpful tool like OOCG should be configured in such a way that the meaning and theme of the game and of its content is conveyed through it. The content in a game is not just randomized pieces but rather carefully placed pieces in carefully chosen positions. If a platform is unreachable in a platform game is has to have a purpose other than being there for the player character to walk on. It is not hard to imagine the frustration of a player trying to reach a reward that is out of reach. Level design is perhaps a form of art because of the balance between providing challenges and rewards. A challenge that is too simple or too makes the player bored or frustrated. The essence is to provide the content in a way that a player can rise up to challenges and barely make them. In state-of-the-art game development this is solved in the painstaking process of play testing and iterative design. It would therefore be interesting to try to utilize previous play testing efforts and a possible way of doing this is to be able to read a previous level design and generate new content from this in such a way that the meaning of the content is not lost but still new.

DESIGN PATTERNS IN GAMES

In games, design patterns can be seen as providing answers to problems faced by the game and/or level designer. However, we can also see each pattern as a problem posed by the designer to the player. If we view the content in a level as challenges or problems the player must find a solution in order to continue the progression through the game. The idea of automating the construction of problems with a given solution is a strategy to avoid the limitations of constraint checking, allowing content to be produced without creating impassable obstacles for the player. Furthermore, if we use previously play-tested problems we can with some certainty reuse problems that are setting different skill sets and skill levels and thus provide more appropriate content for particular players or player types. The seminal work of Bjork and Holopainen introduced design patterns to game design, and provides the foundations for the contemporary discussion about the topic. The book describes hundreds of design patterns, at different levels of abstraction and with reference to different game genres and tasks of game design. Here we will focus on patterns in the design of game levels (and similar spatial designs, e.g. maps and tracks) as opposed to e.g. patterns in game user interface design or rewards. Hullett recently analysed levels of a common first-person shooter (FPS) game in order to and recurring design patterns that had an impact on game-play. The patterns the found include arenas, sniper positions and galleries, commonly seen in many FPS games. The work that is most closely related to our current concern is the work already done on design patterns in platform games. Smith et al. analysed the design of platform game levels, and later devised the Tanagra mixed-initiative level generator. Tanagra uses a constraint solver to generate level geometry in interaction with a human designer. The geometry is generated according to a number of patterns. These patterns occur at two different levels, the single-beat (micro) level where patterns such as the "gap pattern" and "spring pattern" can be found, and at a slightly higher level, where patterns such as "valley" and "mesa" are composed of three micro-patterns each. These patterns are implemented with some exhibility, as the constraint solver can decide to stretch them to some extent to _t in the overall structure of the level. Peter Mawhorter describes a level generator for Super Mario Bros based on "occupancy-regulated extension" (ORE). The generator works by connecting a number of small level chunks like pieces of a puzzle, and generates levels of more novelty and interestingness than many comparable level generators. ORE could be seen as a compositional approach to implementing design patterns in procedural level generation. However, the description of ORE which can be found in the literature only allows for small and static (non-parameterised) level chunks.

ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785
(ISRA), Journal Impact Factor: 2.114

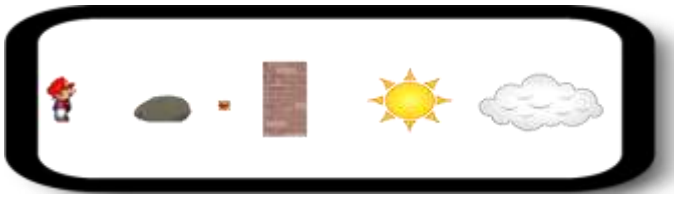
COMBINING OOCG AND DESIGN PATTERNS

OOCG and design patterns could plausibly be combined in several different ways, even when limiting the context to level design. Perhaps the most straightforward way is creating compositional content generators that view each pattern as a spatial design element and simply combine these elements by connecting them next to each other. This can be done either with static elements, as in Mawhorter's ORE, or with parameterized elements, as in Tanagra. Patterns could be connected sequentially in one dimension (as in Tanagra), two dimensions (as in ORE) or potentially in three dimensions. It is also conceivable to stack patterns, i.e. place several patterns at the same place. This would have the effect of modulating one pattern by another. Some patterns may fit well to certain patterns if it is placed before or after. Another way of using patterns in the OOCG process is to use patterns as objectives, e.g. as evaluation/fitness functions or constraints in search-based OOCG. The existence of particular patterns could be seen as desirable or undesirable properties, biasing the search in content space so that the resulting content would be more likely to include or not include certain patterns. Such an approach could potentially lead to more variation than the composition-based approach, but is also more computationally expensive and harder to predict. An example of this approach is the “choke point" evaluation function in a recent attempt to evolve maps for the StarCraft real-time strategy game. Maps which contain choke points are assigned higher fitness and the results of the level generator are therefore likely to contain this particular pattern. With both approaches, patterns can be selected that are particularly well suited to a particular player, for example in order to maximise entertainment as predicted by a player model.

COMPONENTS

Any screenshot of a game in play reveals certain elements that appear consistently and re-occur. This are Generic Design Elements called Components. They are the fundamental units of Level. Every object on the screen can be considered as a Component.

They can be split into two categories, Dynamic and Static. Dynamic components have user interactive functionalities. Ex: Boulders, Obstacles, Coins, Enemies etc. Static Components are the non-interactive elements such as clouds, bushes etc.



A sample component class may have two parameters of importance – its position(x,y co-ordinates) and Type(the type of character it is)

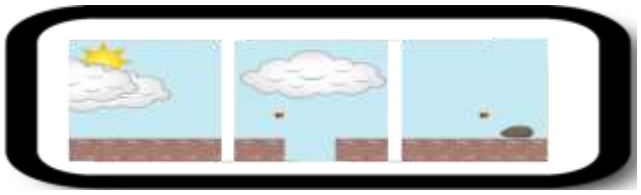
```
public class Component {

        private int x, y;

        public Component(){
        }

        /**
         * Initializes a new component with specified location
         */

        public Component (int x, int y) {
                this.x = x;
                this.y = y;
        }

        public int getX() {
                return x;
```

ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785
(ISRA), Journal Impact Factor: 2.114

```
        }

        public int getY() {
                return y;
        }

        public void setX(int x) {
                this.x = x;
        }

        public void setY(int y) {
                this.y = y;
        }

        public void setPosition(int x, int y) {
                this.x = x;
                this.y = y;
        }
}
```

VIEWS

A view a single frame of the level that is visible at any point during the gameplay. It is a modular holding components. The arrangement of components in a view is pre-defined/manually created by the designer.
A variety of Views can be created depending on the kinds of sections on wants to create. Since this entity is a Static one and not Dynamic, the designer is advised to create a large variety of such Views in order to enable the levels to be interesting.



The Views also have a parameter called Difficulty. The difficulty has to be manually assigned by the designer upon creation of the view; it is of critical importance later on.

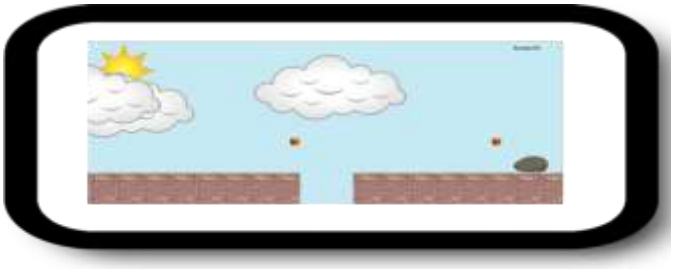
```
public class View extends Vector<Component> {

        private int height, width, difficulty;

        public View() {
                super(1, 1);
        }

        public void setDifficulty(int x) {
                difficulty = x;
        }

        public int getDifficulty() {
                return difficulty;
```

ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785
(ISRA), Journal Impact Factor: 2.114

```
        }

        public void setHeight(int x) {
                height = x;
        }

        public int getHeight() {
                return height;
        }

        public void setWidth(int x) {
                width = x;
        }

        public int getWidth() {
                return width;
        }
}
```

PATTERNS

A pattern is an ordered collection of views appearing sequentially. It defines the layout of a section of a level. The number of views constituting a pattern varies and depends on user preference. The specific order in which the views appear is crucial as it would determine the layout.
A pattern also has a Difficulty parameter which is computed as the average of the difficulties of all constituent views.



A Sample Pattern class may look like this –

```
public class Pattern extends Vector<View> {

        public Pattern() {
                super(1, 1);
        }

        /**
         * Returns Difficulty
         */

        public int getDifficulty() {
                int sum = 0;

                for (View v : this)
                        sum += v.getDifficulty();
```

ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785
(ISRA), Journal Impact Factor: 2.114

```
                return sum / size();
        }
}
```

LEVEL

A level is an independent unit representing a Single level. It holds the entire content of one level in the form of consecutive views/frames.
Save/load functionality can be added to this class in order to save/store a level for subsequent use later.

LEVEL GENERATOR

This is the class where the bulk of the computation is done. It works out the actual layout of the level. It obtains patterns from the repository, selects the appropriate ones depending on the Difficulty and then arranges them. Then the individual patterns are split into their component Views. The end result is an infinitely long sequence of randomly ordered Views.
This can be of two types, Dynamic and Static

Dynamic Generator-
Generates the layout in Real-Time. It selects one view at random and returns it the ongoing level as the next frame. Used while playing the game live.

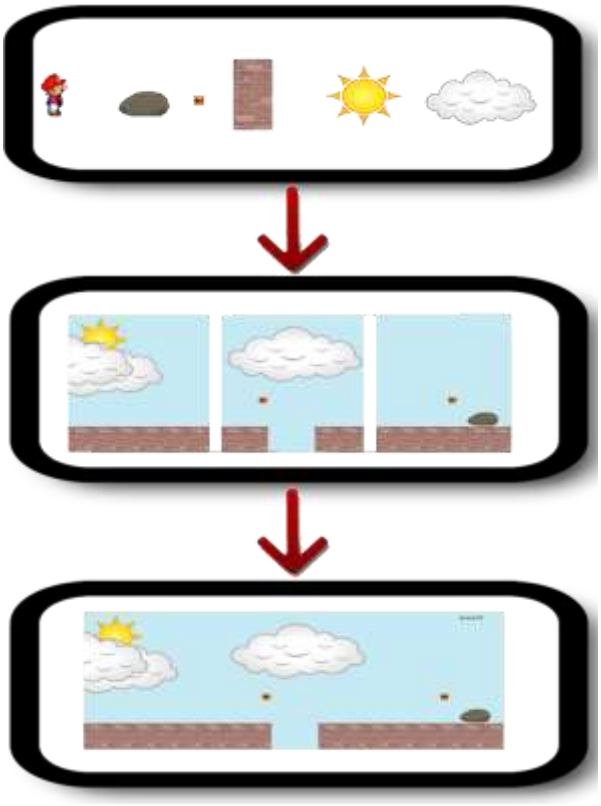
Static Generator-
Generates the layout Statically. The entire level is generated in one go. NOT in real-time. Used to generate and store levels for subsequent use.

```
public class LevelGenerator extends Vector<Pattern> {

        private int difficulty;
        protected Random rand;

        public LevelGenerator(int difficulty) {
                super(1, 1);
                this.difficulty = difficulty;
                this.rand = new Random();
        }

        /**
         * Only Adds Patterns with matching Difficulty
         */
        @Override
        public boolean add(Pattern p) {
                if (p.getDifficulty() != difficulty)
                        return false;

                return super.add(p);
        }

        public int getDifficulty() {
                return difficulty;
        }

        public void setDifficulty(int difficulty) {
                this.difficulty = difficulty;
```

ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785
(ISRA), Journal Impact Factor: 2.114

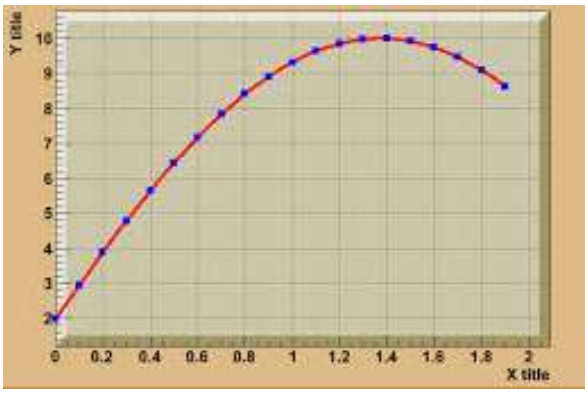
```
        }
}
```

HIERARCHY



DIFFICULTY

Difficulty parameter as described earlier is an important factor in designing of levels. As the content changes, the difficult should also be altered to keep the player in the flow of the game. The following difficulty graph will explain this concept clearly.



ISSN: 2277-9655
(I2OR), Publication Impact Factor: 3.785
(ISRA), Journal Impact Factor: 2.114

The essence of the game is decided by its difficulty. The game should be hard enough for the player to put in sufficient effort but not so hard as to de-motivate him. The difficulty should be just 'Out of Grasp'. The designer can choose the 'feel' of a level by altering its difficulty graph. Easier/Harder levels can be decided by altering the graph shape.

CONCLUSION

In this paper we have discussed the potential roles of design patterns in OOCG and presented a Universal Level Generator utilising these concepts. By ordering the patterns in sequence of difficulty we can vary the content according to what a player does.

REFERENCES

1. Steve Dahlskog , Julian Togelius - Procedural pattern generation.
2. C. Alexander, S. Ishikawa, and M. Silverstein – A pattern language , Oxford University Press, New York, U.S.A., 1977.
3. S. Bjork and J. Holopainen - Patterns in game design. Cengage Learning, 2005.
4. Apostolos Ampatzoglou and Alexander Chatzigeorgiou - Evaluation of object-oriented design patterns in game development, 2006
5. K. Compton and M. Mateas - Procedural level design for platform games i n Proceedings of the 2nd Artificial Intelligence, 2006
6. Divyansh Prakash - 3Coffee Game Engine: https://github.com/divs1210/3Coffee
7. Ben Weber - A Probabilistic Multi-Pass Level Generator, 2010