Building a fire propagation system in real-time graphics

Kristian Olsson

Computer Graphic Arts, bachelor's level
2017

Luleå University of Technology
Department of Arts, Communication and Education
Preface

This thesis concludes my studies at Luleå University of Technology, in the subject of Computer Graphic Arts. I would like to thank the university and its personnel for providing knowledge and insight about the profession and industry. This bachelor thesis was written at Pixelgruvan, thank you for your guidance and support.

Special thanks to my family and friends who have helped and supported me through this journey.

_Kristian Olsson_
Abstract

This report covers the creation of a dynamic fire propagation method for a real-time environment. The purpose is to see if it is possible to create a system that can control fire propagation behaviour and visual design based on some sort of simple parametrization, the purpose stems from the lack of a system to control and design a fire propagation scenario. To attain the results, a fire propagation method is devised based on the purpose of having a parameter based system, this method is created through the use of scripting in a real-time game engine to control visuals and behaviour of built in particle systems. Results show fire propagation through an example scenario where the fire behaves differently based on the material that is burning, based on parameters set by an artist. These results conclude that is it possible to create a parameter based fire propagation system and that it can be used to change the visual design and behaviour and be expanded to provide better artist input and control. The report suggests further research in the area of simplified controlled fire simulation in real-time engines, and usability.

Sammanfattning
Table of Contents

1. Introduction & background 1
   1.1 Purpose 2
   1.2 Question at issue 2
   1.3 Limitations 2

2. Method 3
   2.1 Approach 3
   2.2 Visualization method 3
   2.3 Fire propagation pillars 4
      2.3.1 Fire propagation environment 4
      2.3.2 Master script 5
      2.3.3 Individual particle script 5
   2.4 System architecture 6
   2.5 Method Critique 8

3. Result 9

4. Discussion 11
   4.1 Result 11
      4.1.2 Validation 11
      4.1.3 Further research 12
   4.2 Method 12

5. Conclusion 13

6. References 14

7. Appendix 15
   7.1 Appendix A 15
   7.2 Appendix B 18
   7.3 Appendix C 26
1. **Introduction & background**

Real time graphics have been around for a long time, and as it continues to evolve, so does the area of use. Along with sophisticated real-time graphics engines comes the will to try to simulate real life scenarios. Scenarios where one wants to put the observer into the world you are creating, where they can feel like they are immersed in the world. A part of that experience is the feeling of plausibility, therefore there is a need to create methods that make something feel real.

In the visualization industry there is always a need to design and present scenarios in different ways. Sometimes a real-time approach is used, where one would want to adapt and change the scenario in a dynamic way. E.g. If the changing behaviour of fire is something one want to visualize, one would need to quantify what it is that makes it behave the way it does and then visualize that in some way. Similar studies have been conducted by Lever and Komura (2012) where the altering of a fire behaviour and look has been a central research subject.

As virtual reality is getting increasingly popular and bigger, there is now a possibility to move certain real-time scenarios into the virtual world instead of the real one, this can bring down costs for training personnel for different jobs or be used for safety educational purposes.

Nicklas Karlsson (2017), trained fire safety and risk consultant with CFPA certification at Innan AB explains in an interview the need for this type of virtual scenarios.

> The medium for safety education and training have not changed so much since the 1980’s. Training and educate people with existing methods have very low retention due to lack of realism & engagement by participants. The allocation for a realistic environment in safety educations is very demanding and sometime impossible to conduct. We see the need to innovate the medium being used, by providing a virtual reality immersive experience, where people could be put into risk environments in a controlled and safe way. This will combine practical application with theory in a virtual reality scenario, giving increased understanding, engagement and retention of different high risk scenarios (Karlsson, 2017).

Therefore there is a need to investigate how to create such scenarios, a risk fire propagation scenario requires systems that can simulate an advanced scenario simplified in real-time. There are many different parameters that are interesting in how one can control fire propagation, and how these parameters affect how fire propagation look. According to Svenskt Trä (2016) we see that the time before wood ignite is based on the level of moisture in the material and the incoming heat from a heat source. We can assume a simplified quantification for this to be that if something is going to burn, it has resistance to heat, and fuel in the form of material that has not reached its ignition point. Furthermore, these
parameters also gives two other parameters, different states that says that the material is burning or is not burning. These simplified parameters will be used as a base to create a simplified fire propagation method.

1.1 Purpose
There are real world scenarios where one would want to use some sort of visual design of a fire propagation scenario. E.g. a virtual reality firefighter training simulator where fire has to look and behave as real as possible. This means that one would want to control fire behaviour and visual look with some sort of parametrization.
The report will focus on developing a method to visualize a fire propagation in a real-time engine, and how one with simplified parameters can control the fire propagation.

1.2 Question at issue
How does one simulate and design fire propagation in a real-time engine?
- Are there any existing systems for simulating fire propagation in a real-time engine? If there is, is there an alternative approach? If no method exists, how does one proceed with doing this?
- How can one use simplified parameters in 3D, to design fire propagation?

1.3 Limitations
This report covers the question at issue in the field of computer graphics. The study focuses on providing a method based on knowledge suitable for a technical artist. The report will focus on how to create a system to design fire propagation in a real-time engine. And will be specified for that purpose. The report will focus on the parametrization of a fire scenario. The report will cover a method that is meant to be used specifically for fire and will not go into detail about other scenarios where the same method might be applicable.
The report will focus on 3 example materials for the fire to spread to, this is to limit the scenario to a manageable level in the report. What these materials are will not be specified, their purpose is to show the results of parameters for 3 materials and their properties are not equivalent to real materials, a more in depth separate study will be needed to conclude all the different properties of a material. The report will focus on horizontal spread and will not account for how vertical spread, incline or wind may affect fire propagation, this is since the system will be in an early state and the focus will be on the question at issue during the duration of the study. These features can be implemented in an expanded system outside of this report. The visual results of the fire propagation will be limited since the system will be in an early state, therefore the report will describe the means to create a fire propagation system. If the system works, then one can expand it to give better visual control and results.
2. Method

2.1 Approach
A method will be devised based on empirical background in the subject. In this case using programming to develop a method to solve the question at issue, based on the requirement of having a parametric based system. The system utilizes parameters that are based on information gathered in the background study.
Theoretically transforms in a scene can act as spawners for particle systems that is used to create an effect that looks like fire.
To solve the spawning of fire, there is a need for a fire propagation system.
The fire propagation system should use different subsystems that handle different parts of the fire propagation scenario. The systems could be: A code that handles the spawning of particle systems and keeping track of different transforms in the scene. A code that controls individual particles and their attributes and help them talk to each other. There is also a need for a list of material data which the particle systems can use to know how to behave and look. These are the aspects of which the propagation system is built upon.

2.2 Visualization method
Since we can not use full fledged, highly accurate simulations that can deal with fire propagation in real-time as the time of this thesis creation, one has to use simplified methods to simulate fire propagation. This needs to be done while still maintaining some of the accuracy of real fire propagation.
The chosen method is using Unity3d engine V5.6.0f3 which is a real-time game engine. The method provided here is based on scripting in C# as explained by Wagner, Latham and Wenzel (2017) and Json as explained by Ecma International (2013). The Json implementation is using a dll extension called LITJSON explained in Lbvgithub.io (2017).
Since the approach used is using code to drive a particle system, this can be applied to other engines as well, or a system built from the ground up. Therefore the choice to use Unity engine should not have any major implications when this method is being used in other real-time systems. A built in particle system is used since major real-time engines use a system of this kind.

In a 3D game, most characters, props and scenery elements are represented as meshes, while a 2D game uses sprites for these purposes. Meshes and sprites are the ideal way to depict “solid” objects with a well-defined shape. There are other entities in games, however, that are fluid and intangible in nature and consequently difficult to portray using meshes or sprites. For effects like moving liquids, smoke, clouds, flames and magic spells, a different approach to graphics known as particle systems can be used to capture the inherent fluidity and energy. (Unity Technologies, 2017)

Particle systems by themselves might differ, but the the same method can be used as a guide to achieve the same results. Unity3d engine is using their own particle system called Shuriken Unity Technologies (2017), which will be used in this method.
2.3 Fire propagation pillars
To create a system that can handle fire propagation there needs to be some basic pillars for the system to be built upon. There needs to be a scene, which there is by default in Unity where the particle system can propagate. For this method to work there are transforms scattered around in a partially fixed, and partially random pattern. The transforms have an X, Y and Z position in 3D space along with a name. The locations of the transforms can be set depending on how one wants to build their system, the system used in this method is made for testing purposes and might not be suitable for every scenario.
An important aspect of creating a script that is handling particle systems is how objects are handled in the real-time engine. Objects in unity are put into GameObjects according to Unity Technologies (2017). GameObjects are essentially empty boxes which can be filled with different components. When there is a need to access something in the gameObject the code calls a gameObject and then finds the content. Particle systems can be put into GameObjects as a component. The use of GameObjects are favorable in this method since a lot of different data can be stored there and found (Unity Technologies, 2017).

2.3.1 Fire propagation environment
The creation and placement of transforms is one of the ground pillars for this system to work, the placement and naming of the transforms can be scripted and created in a 3D modeling software where one could theoretically spawn transforms all around in a scene and depending on if their position is inside of an object that has a certain material, they would be created and given a position and a certain identifier.

The transforms for this method however are hand placed and named so that they include an identifier that indicate what kind of material it is. Including a identifier makes it so that the identifier can be used to set material specific parameters later in a script.
See Figure 2.1 where the bold section of the text is the identifier for the transform.

Figure 2.2 shows locators that are placed in a 3D modeling software scene before they are imported into unity as transforms.
The main purpose of the transforms is to act as guides so the script knows on which positions to spawn fire particles and how they should burn. The transforms are put into GameObjects in Unity.
2.3.2 Master script
To keep track of particles in the scene there is a “ParticleMaster.cs” script. This ParticleMaster script handles a number of different tasks.
- Creating a particlesystem in the real-time engine based on a particle system preset.
- Instantiate the particle system on different locations in the scene based on the transforms in the scene.
- The script has lists containing data about: If particles in the scene are in a is burning state, all the particles that are in the scene and if particles have burnt out.
- The script acts as a database to which every particle system can pull and push data and act according to certain rules.

The particles systems in the scene are assigned a script that in this method is called a “SpreadScript.cs”, that can change parameters for each individual particle system. This is done by applying the script on the particles preset so that when the ParticleMaster script instantiates the GameObject the particle system are assigned to, all of the systems are assigned an individual script.

2.3.3 Individual particle script
The SpreadScript scripts are going to be running simultaneously on many particle systems so the script in general should be made to be lightweight, therefore making as few as possible calculations and creations in the script is favourable. The most optimal way of handling this is to have the script read data, and change as few variables as possible.
In the script for this method there are some calculations going on which impact the performance of the fire propagation, this can be optimized by designing the lookup for which particles are burning differently.

The SpreadScript handles individual particle system calculations and checks, which include:
- Setting initial attributes for the individual particle systems based on the name identifier of the transforms in the scene.
- Start and stop updates of the individual particle systems so that the real-time engine does not have to calculate particle systems that are inactive.
- Run checks that provide information about if the particle system should be enabled, and get a is burning state. E.g. If a neighbouring particle system is burning and is close enough, this particle system should be burning.
- Run checks that tell if the particle system should stop burning or not, depending on different parameters set. E.g. fuel amount left, outside forces such as fire extinguisher.
Figure 2.3 is showing a list of how the scripts are applied to the GameObjects. The ParticleMaster is applied to the Master_GameObject_01 GameObject, and then when the game engine is running, it creates the GameObject_Transform_01_ParticleSystem GameObject, applies the SpreadScript and then instantiates it.

The “matAttr.json” script is set up as a list of different presets that provide information to the SpreadScript about what kind of material the particle systems should use to determine their starting values, the type of values are based on information obtained in the background study, the numbers are fictional however.

In Figure 2.4 we see an example of such a list, where the value of “id” should correspond to a transform identifier, and if a transform has an identifier that is not in the matAttr list they will use set default values. This list is what can then be used to design how the fire propagation should look and behave depending on the material in the scene. The list can be expanded to include different attributes E.g. fire color, particle amount and particle speed.

2.4 System architecture
Scripts running in a real-time engine have different ways of executing its functions.
The standards script layout that is used in this method has two major functions that define how code is executed.
The `start();` function and the `update();` function.

The `start();` function runs when the scene being worked on is initializing play mode. While in play mode the real-time engine starts calling the `update();` function once every frame. It is important to distinguish these two functions since they handle different tasks and will be used for their respective purpose in the scripts. `start();` is used as an initializer of data, and `update();` is an updater that can recognize changes in a scene during runtime.

The fire propagation system works by executing several steps. In Figure 2.5 there is a flowchart showing the execution of the scripts in runtime.

1. The ParticleMaster script initiates its `start();` function and creates the SpreadScript scripts, place them on the location of the transforms in the scene and give them the name identifier of the transform they spawn on.
2. The ParticleMaster script creates two dictionaries. One is used to determine if particles are burning, a “isBurningDict” dictionary. And a “burntOutDict” dictionary that gathers particles that has burnt out. These two are used by SpreadScript scripts to determine if they should burn or not.
3. The SpreadScript script is initialized and initiates its `start();` function, it gathers data from the matAttr.json file and give the particle system its initial attributes from the file.
4. The scripts run their `update();` function. This makes them go into update mode and scan for changes in the scene. As long as nothing in the scene is reporting “i am burning” nothing will happen.
5. The user or by a script, enable emission on one or several particle systems in the scene.
6. The SpreadScript scripts applied to the particle systems that are burning sets a “is burning” state. And send that to the isBurningDict dictionary. This results in giving a list that has all the particle systems that are currently burning.
7. The particle systems in the scene that are not burning start comparing their position and distance to the particle systems that are burning. If the burning particle systems are close enough they affect the non burning particle system so that it ignites, and send itself to the isBurningDict dictionary.
8. Particle systems that are burning checks for changes in its fuel amount and things in the scene that will affect its attributes.
9. Particle systems that are out of fuel stops emitting and are sent to burntOutDict dictionary where they in the ParticleMaster script gets removed from the isBurningDict.
10. Step 6-9 repeats until every particle system in the scene has burnt out.
2.5 Method Critique
The method chosen to determine "How does one simulate and design fire propagation in a real-time engine?" requires a lot of setup with a lot of step by step progression based testing to make it work. Therefore it could be very time consuming to replicate, especially if knowledge of programming is limited. The data acquired from the results may not give an accurate answer to the question at issue, this is since to get a result that is visually accurate and have an extensive impact on the behaviour one would need to expand the code and include several more parameters that would change the behaviour and looks of the fire. That would however require extensive time and personnel. The core functionality of the code and simple test parameters should answer if it is technically possible, although it might not suffice for a visual validation.
3. **Result**

Results are showing the code running in Unity3d engine V5.6.0f3 where it affects the particle systems. The code results are available in appendix A, B and C.

**Figure 3.1**

*Figure 3.1 Shows fire propagation through 3 different materials in a real-time environment. These are progression snapshots from 5 to 15 seconds.*

**Figure 3.2**

*Figure 3.2 Shows fire propagation through 3 different materials in a real-time*
environment. These are progression snapshots from 20 to 30 seconds.

*Figure 3.3* Shows fire propagation through 3 different materials in a real-time environment. These are progression snapshots from 35 to 55 seconds.

*Figure 3.4* Shows fire propagation through 3 different materials in a real-time environment. These are progression snapshots from 1 min 20 sec to 1 min 32 sec.
4. Discussion

The purpose of the study was to determine: How does one simulate and design fire propagation in a real-time engine?
- Are there any existing systems for simulating fire propagation in a real-time engine? If there is, is there an alternative approach? If no method exists, how does one proceed with doing this?
- How can one use simplified parameters in 3D, to design fire propagation?

4.1 Result

From conducted research there were few existing studies found that revolves around fire propagation in real-time. There are a few computer games that have fire propagation systems but there are no studies or papers available for them to make an assessment about if there are similar systems and how they work. Results are showing that the fire propagation method is working, the fire is spreading throughout an example scene and is doing this with the help of custom parameters set by a user. This gives a result showing that it is working in code and shows that a dynamic fire propagation can be changed by input from artists, and that it is not an effect that has to be animated but instead can act differently every time the real-time engine is running the scene.

From the results we see that the fire starts from the left side and spreads until arriving to the right, since the materials have been given different attributes by an artist, the fire behaves differently depending on the material and on where the fire starts in the scene, therefore confirming that it is possible to use simplified parameters to design fire propagation, the parameters used are based on the parameters gathered from background study.

Though this is done by simple parameters one could extend this to change size, color and other settings to improve the possibilities of different visual results. For the question at issue this is what is desirable.

The results confirm that it is possible to analyze real material aspects as is done in background study, convert them into simplified parameters and implement them into a system that use them to control the fire propagation behaviour and look.

4.1.2 Validation

Concerning the validity of the study, the results here are showing that it is working in code and one can create a propagation system based on parameters, although it is very difficult to determine visually from an artistic point of view if the method is suitable for getting the look right.

Taking into account several aspects such as, to what extent can one change the fire through parameters alone, should it look and behave like real fire or stylized. The answers to these would be, it depends. If one wants to create fire propagation that looks real, one would need to analyze different materials and quantify the burning process, take the data gathered and translate it into parameters that one can put into the attribute list and then alter the behaviour and looks by code. Whereas if one wants to create a stylized fire one might want to have a different behaviour and have to come up with parameters that would suit those stylized needs.
Creating the desired look with parameters could prove difficult from an artist standpoint where one is bound to how the system is build, more so in this method since the particle system must work in the fire propagation system. Normally one would animate the particle system, and tweak the values in the particle system from the visual feedback, it is easier to predict the visual outcome in a designed system. In the method displayed here the result is different every time since there is randomness implemented in the system, which will make the design of the visuals more difficult for the artists.
Something that speaks highly for the fire propagation system is that if there is a scene with hundreds of particle systems an artist does not have to go in and change the look for all of the individual systems. The artist can create a new material in the attribute list provided and then say which particles that should have that material, this would speed up the process of look development in a massive fire propagation system.

4.1.3 Further research
The method and results bring up new questions about how one could use this kind of system and evolve this system from a virtual education and safety standpoint. Real time engines have mostly been used for games in the past, but with vr becoming more available, further research should include what one can do with it in scenarios outside of games. Interesting areas to branch out to is if a better result can be achieved with techniques other than particle systems and how one can quantify how materials burn and then translate that into a real-time environment and simple simulation, creating more interesting and complex visual results. The lack of similar studies show that there is need for more research in the subject of simplified real-time fire propagation simulations where we want artist input and control over material properties and visuals.

4.2 Method
Using an empirical experimental method is not favorable in the sense that it is difficult to assess what you need to make the system work in the beginning. There is a lot of progression based research with trial and error, therefore it takes a long time to get any usable results even if one has a plan for what the end results should be like. When the system is in an operable state however, it is quite fast to add and improve the code so that there is a better visual result.

There is a lot of improvements that could be made to improve the results, it could be optimized so that one can get a denser amount of transforms in the scene. Which would result in a smoother fire propagation and give a more realistic and visually correct result where the flames would spread gradually, the further apart the transforms are the more popping will occur since the particles spawn on the locations of the transforms. Another way to go is to get the system to be able to run threaded, or run it on the GPU as separate calculations instead of the CPU as it does right now, but that would require rewriting a majority of the system. Since this method is built with a kind of modularity in mind, it is quite easy to add increased complexity. Such as different fire propagation speed up and downwards, add wind forces and make the fire propagation respond to it and give more visual feedback.
At the same time when adding more complexity one adds more calculations which in turn slows down the system, bringing it closer and closer to non real-time simulation which is the opposite of what one wants in this method. The method used shows that it is difficult to design a system that is easy to use by an artist, the method is very technical and there was too much focus and time spent on the technical aspects so the visual result is not as refined as it could be.

5. Conclusion

It is possible to create a parameter based fire propagation system based on the method in this report. The results show that the system is in its cradle but that it is a valid method to simulate and design fire propagation in a real-time engine.

It shows that a simplified parameter system provides a way to control the visual design of fire propagation in a very material specific way. Using a parameter based system can have major benefits in large scenes that are complex to design manually. Artist input and control over the propagation system prove to be limited since a lot of development is needed to provide more visual based parameters.

The report highlights that there are complexities in creating a parameter based fire system, while a dynamic system create interesting results and different results every time, it is more difficult to design compared to a static system.
6. References


7. Appendix

7.1 Appendix A
Appendix includes source code for ParticleMaster.cs script

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ParticleMaster : MonoBehaviour
{
    public Transform rootObject;
    public List<GameObject> gameObjectsList = new List<GameObject>();
    public Dictionary<GameObject, bool> isBurningDict = new Dictionary<GameObject, bool>();
    public Dictionary<GameObject, bool> burntOutDict = new Dictionary<GameObject, bool>();
    public Dictionary<GameObject, bool> burntOutDictCurrent = new Dictionary<GameObject, bool>();
    public Dictionary<GameObject, bool> isBurningDictCurrent = new Dictionary<GameObject, bool>();

    float time = 0f;
    //float timePassed = 0f;

    float vecDist;
    Vector3 listAddVector;

    // Time interval every 1 seconds
    // int interval = 1;
    float timeNow = 0f;

    public GameObject prefabElement;

    // Use this for initialization
    void Start()
    {
        gameObjectsIntoList();
    }

    // Update is called once per frame
    void Update()
    {
        timeUpdate();
    }

    //timer for updatetick
    void timeUpdate()
    {
        // Time counter
```
time += Time.deltaTime;
// Run every second
if (Time.time >= timeNow)
{
    isburningIntoDict();
burnOutIntoDict();
    // print("key count isburninglist " + isBurningDict.Keys.Count);
timeNow += Random.Range(0.10f, 1.00f);
}
} //Function for adding particlesystems into list. Call (gameObjectsList) for list
void gameObjectsIntoList()
{
    foreach (Transform getObject in rootObject)
    {
        GameObject fireInstance = Instantiate(prefabElement, getObject.position,
Quaternion.identity);
        fireInstance.name = getObject.name + "_instParticle";
        gameObjectsList.Add(fireInstance);
    }
    /*Loop to assign instances as children under rootObject.
    this is done outside creation loop above since Unity crashes when assigning a
    parent in the creation loop.
    */
    foreach (GameObject getObject2 in gameObjectsList)
    {
        getObject2.transform.SetParent(rootObject, true);
    }
} //Function for creating a dictionary that says if a particlesystem is active or not. call
(isIsBurningDict) for dictionary
void isburningIntoDict()
{
    ParticleSystem checkParticleActive;

    foreach (GameObject gObj in gameObjectsList)
    {
        checkParticleActive = gObj.GetComponent<ParticleSystem>();
        ParticleSystem.EmissionModule PSysCheck = checkParticleActive.emission;
        if (PSysCheck.enabled == true && !isIsBurningDict.ContainsKey(gObj))
        {
            isIsBurningDict.Add(gObj, true);
        }
        else
        {
        }
    }
} //Function that adds burnt out fires to a dictionary and removes them from
is BurningDict dictionary
    void burnOutIntoDict()
    {
        // print("burnOutIntoDict running");
        if (burntOutDict.ContainsValue(true))
        {
            burntOutDictCurrent = burntOutDict;
            isBurningDictCurrent = isBurningDict;

            foreach (GameObject gObjTmp in burntOutDictCurrent.Keys)
            {
                if (isBurningDictCurrent.ContainsKey(gObjTmp))
                {
                    isBurningDictCurrent.Remove(gObjTmp);
                }
                else
                {
                }
            }

        } 
        isBurningDict = isBurningDictCurrent;
        isBurningDictCurrent.Clear();
        burntOutDictCurrent.Clear();
    }
    else { }
Section 7.2: Appendix B
Appendix includes source code for SpreadScript.cs script

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO;
using Newtonsoft.Json;

public class SpreadScript : MonoBehaviour
{
    public ParticleParameter fromMaster;
    Transform ignitionPoint;
    Dictionary<GameObject, bool> isBurningDict = new Dictionary<GameObject, bool>();
    Dictionary<GameObject, bool> burntOutDict = new Dictionary<GameObject, bool>();
    Vector3 currFirePos;
    private GameObject currObj;
    JsonData jsonDataIn;
    string textImport;
    string itemToText;

    // Default burnstate, if current GameObject is burning, and if the GameObject has burnt out.
    bool isBurning = false;
    bool burntOut = false;

    // Initialize properties.
    //----------------------------------------
    string[] nameArr;
    //string materialType;
    int materialID;
    string materialType;
    float fireFuel;
    float fuelVariance;
    float resistHP;
    float resistBurnRate;
    float resistBurnRateMult;
    float pScale;
    float pScaleMult;
    float pBurnRateMult;
    float pBurnRate;
    float pEmissionRate;

    //----------------------------------------
    float time = 0f;
    float timePassed = 0f;
```
// Time interval every 1 seconds
// float interval = 0.5f;
float timeNow = 0f;
float vecDist;
Vector3 listAddVector;
bool found;
float spreadDistance = 0.5f;

// Use this for initialization
void Start ()
{
    fromMaster = GetComponentInParent<ParticleMaster>();
    is BurningDict = fromMaster.is BurningDict;
    burntOutDict = fromMaster.burntOutDict;
    currObj = this.gameObject;

    pSystemStopUpdate ();
    StartCoroutine (loadText ());
    StopCoroutine (loadText ());
    streamDataFromJson ();
    SetSpecialAttributes ();
}

// Update is called once per frame
void Update ()
{
    // Invoke("timeUpdate", Random.Range(0.001f, 1.000f));
    timeUpdate();
    //
}

// timer for updatetick
void timeUpdate()
{
    // Time counter
    time += Time.deltaTime;
    if (Time.time >= timeNow)
    {

        if (getCurrentBurning() == true)
        {
            // run check commands againts struct to see if fire should fade or change behaviour.
            fireOut();
        }
        else
        {
            if (burntOut == false && is BurningDict.Keys.Count <= 400) {

                startFire ();
            } else {

            }
        }
    }
}
} 

} 

timePassed += 1;
timeNow += Random.Range(0.10f, 1.00f); 
} 
} 

// Function that checks if that current gameobject is in the is currently burning list 
bool getCurrentBurning() 
{ 
    try 
    { 
        if (isBurningDict.TryGetValue(currObj, out found) && found == true) 
        { 
            // print("Object IS in burnlist"); 
            isBurning = true; 
            return true; 
        } 
        else 
        { 
            // print("Object is NOT in burnlist"); 
            isBurning = false; 
            return false; 
        } 
    } 
    catch 
    { 
        return false; 
    } 
} 

// Function gets the type of material currently applied to the transforms (material attribute ID) 
void attrFromTransform() 
{ 
    nameArr = currObj.transform.name.Split("_"); 
} 

// Function streams data from .JSON file in Resource folder. Checks if the current 
material attribute ID is in the preset. 
void streamDataFromJson() 
{ 
    attrFromTransform(); 

    jsonDataIn = JsonMapper.ToObject(textImport); 

    int arrLength = nameArr.Length; 
    int loopTotal = jsonDataIn ["List"][0].Count * arrLength; 
    int loopTick = 0;
for (int i = 0; i < arrLength; i++)
{
    //print ("array length is " + arrLength);
    for (int x = 0; x < jsonDataIn ["List"].Count; x++)
    {
        if (nameArr [i] == jsonDataIn ["List"] [x] ["id"].ToString ()
        {
            //materialType = jsonDataIn ["List"] [x] ["id"].ToString ();
            materialID = x;
            setInitialAttributes (materialID);
            //print ("Material type is " + materialID + 
            materialType);
            return;
        }
        else
        {
            if (loopTick == loopTotal)
            {
                print ("Material was not in list or 
            attribute in transform name is not 'defaultMat'");
                print ("USING DEFAULT VALUES");
                setInitialAttributesDefault ();
                return;
            }
            loopTick++;
        }
        loopTick++;
    }
}

//Function that collects all the data from the .JSON file and saves it in memory. fuction is a coroutine ment to run uninterupted until done.
IEnumerator loadText()
{
    string toText;
    TextAsset jsonIniData = Resources.Load("matAttr") as TextAsset;
    toText = jsonIniData.ToString ();
    textImport = toText;
    yield return textImport;
}

//Function sets initial attributes collected from .JSON
void setInitialAttributes(int getMatID)
{
    float fireFuelTemp = float.Parse(jsonDataIn ["List"] [getMatID] 
["fuel"].ToString());
    fuelVariance = float.Parse(jsonDataIn ["List"] [getMatID] 
["fuelVariance"].ToString());
fireFuel = Random.Range (fireFuelTemp - fuelVariance, fireFuelTemp + fuelVariance);

    resistHP = float.Parse(jsonDataIn ["List"][getMatID] ["ToList"] " resistHP");
    resistBurnRate = float.Parse(jsonDataIn ["List"][getMatID] ["ToList"] " resistBurnRate");
    resistBurnRateMult = float.Parse(jsonDataIn ["List"][getMatID] ["ToList"] " resistBurnRateMult");
    pScaleMult = float.Parse(jsonDataIn ["List"][getMatID] ["ToList"] " pScaleMult");
    pBurnRate = float.Parse(jsonDataIn ["List"][getMatID] ["ToList"] " pBurnRate");
    pBurnRateMult = float.Parse(jsonDataIn ["List"][getMatID] ["ToList"] " pBurnRateMult");
    pScaleMult = float.Parse(jsonDataIn ["List"][getMatID] ["ToList"] " pScaleMult");

    // Function sets default attributes if attributes cant be collected from .JSON via transform name identifier.
    void setInitialAttributesDefault()
    {
        fuelVariance = 100;
        fireFuel = Random.Range(1500f-fuelVariance, 1600f+fuelVariance);
        resistHP = Random.Range(600f, 800f);
        resistBurnRate = 1.0f;
        resistBurnRateMult = 1.0f;
        pScaleMult = 10;
        pBurnRate = 10;
        pBurnRateMult = 10.0f;
    }

    // Function that initializes attributes that need special code.
    void SetSpecialAttributes()
    {
        /*
         *  ParticleSystem pSysObjSpecAtt;
         *  pSysObjSpecAtt = currObj.GetComponent<ParticleSystem> ();
         *  pScale = pSysObjSpecAtt.transform.localScale;
         *  pScale = pSysObjSpecAtt.transform.localScale;
        */

        // Initiate fire depending on conditions
        void startFire()
        {
            foreach (GameObject gObj in isBurningDict.Keys)
            {
                // Only goes through list until it finds a value that is below threshold distance. (To potentially lower lookup time!)
                if ((currObj.transform.position - gObj.transform.position).magnitude < fireThreshold)
gObj.transform.position).magnitude <= spreadDistance)
{
    if (resistHP <= 0 && fireFuel > 0)
    {
        pSystemStartUpdate();
        ParticleSystem particleSysObjAct;
        particleSysObjAct = currObj.GetComponent<ParticleSystem>();
        ParticleSystem.EmissionModule PSysEmiss = particleSysObjAct.emission;
        PSysEmiss.enabled = true;
        ParticleSystem[] currObjChildAct;
        currObjChildAct = currObj.GetComponentsInChildren<ParticleSystem>();
        foreach (ParticleSystem PS in currObjChildAct)
        {
            ParticleSystem.EmissionModule PSysEmissChild = PS.emission;
            PSysEmissChild.enabled = true;
        }
    }
    else {}

    resistHP := ((resistBurnRate * resistBurnRateMult) * Time.deltaTime * 10);
    //print("ResistHP " + resistHP);
    //print("tick value " + ((resistBurnRate * resistBurnRateMult) * Time.deltaTime * 10));
    //print("Firefuel " + fireFuel);
    return;
    }
else{}

}
//Function that make the fire go out depending on parameters set.
void fireOut()
{
    //Add conditions that make the fire burn out
    //Variable x := y; etc.
    fireFuel := (pBurnRate * pBurnRateMult) * Time.deltaTime * 10;
    //print("FireFuel = " + fireFuel);

    if(fireFuel <= 0 && isBurning == true && burntOut == false)
    {
        ParticleSystem particleSysObjDeact;
particleSysObjDeact = currObj.GetComponent<ParticleSystem>();
ParticleSystem.EmissionModule PSysEmissDeact =
particleSysObjDeact.emission;
PSysEmissDeact.enabled = false;

ParticleSystem[] currObjChildDeact;
currObjChildDeact = currObj.GetComponentsInChildren<ParticleSystem>();

foreach (ParticleSystem PSdeact in currObjChildDeact)
{
    //print("Found " + PSdeact);
    ParticleSystem.EmissionModule PSysEmissChildDeact =
    PSdeact.emission;
    PSysEmissChildDeact.enabled = false;
    // print("PSysEmissChild.deact.enabled = " + PSysEmissChildDeact.enabled);
}
    pSystemStopUpdate ();
burntOut = true;
if(!burntOutDict.ContainsValue(currObj))
{
    burntOutDict.Add(currObj, burntOut);
}
else
{
    //do nothing
}

else {
    // print("just looping fireOut()");
}

//stops the particlesystem from updating.
void pSystemStopUpdate()
{
   ParticleSystem particleSysObjStop;
    particleSysObjStop = currObj.GetComponent<ParticleSystem> ();
    particleSysObjStop.Stop(true,
ParticleSystemStopBehavior.StopEmitting);
    //particleSysObjStop.Pause(true);
}
// starts updating the particlesystem.
void pSystemStartUpdate()
ParticleSystem particleSysObjUpdate;
particleSysObjUpdate = currObj.GetComponent<ParticleSystem>();
particleSysObjUpdate.Play(true);
7.3 Appendix C
Appendix includes source code for matAttrjson script

```json
{
    "id": "defaultMat",
    "fuelVariance": "100",
    "fuel": "2000",
    "resistHP": "1400",
    "resistBurnRate": "10",
    "resistBurnRateMult": "1.0",
    "scaleMult": "1.0",
    "burnRate": "10",
    "burnRateMult": "10.0"
},
{
    "id": "woodMat",
    "fuelVariance": "100",
    "fuel": "3000",
    "resistHP": "800",
    "resistBurnRate": "10",
    "resistBurnRateMult": "1.0",
    "scaleMult": "1.0",
    "burnRate": "10",
    "burnRateMult": "10.0"
},
{
    "id": "gasMat",
    "fuelVariance": "1",
    "fuel": "20",
    "resistHP": "0",
    "resistBurnRate": "10.0",
    "resistBurnRateMult": "10.0",
    "scaleMult": "1.0",
    "burnRate": "10.0",
    "burnRateMult": "10.0"
},
{
    "id": "grassMat",
    "fuelVariance": "50",
    "fuel": "500",
    "resistHP": "100",
    "resistBurnRate": "10.0",
    "resistBurnRateMult": "10.0",
    "scaleMult": "1.0",
    "burnRate": "10.0",
    "burnRateMult": "10.0"
}
```
},
{
    "id" : "gasolineMat",
    "fuelVariance" : "100",
    "fuel" : "2000",
    "resistHP" : "0",
    "resistBurnRate" : "10.0",
    "resistBurnRateMult" : "10.0",
    "scaleMult" : "1.0",
    "burnRate" : "10.0",
    "burnRateMult" : "10.0"
}
]