Verdonck Judith

WILDFIRE SIMULATION

GRADUATION WORK 2021-2022

Digital Arts and Entertainment

Howest.be

Fig. 1 Surface fire in a pine tree forest, [18].

CONTENTS

ABSTRACT

In order to make visualizations of cardiac excitation more recognizable and easier to understand, a simplified wildfire simulator is built. First, a look at real wildfires is taken, going over their properties and characteristics. Then, already existing simulators are looked into and compared. As the goal is to have a wildfire simulator capable of running fire propagation on a full 3D terrain such as a sphere at runtime, some properties of a wildfire will need to be given up. At the end, tests are conducted to see if the application runs at decent fps and how the impact on fps evolves with higher detail and the addition of VFX.

INTRODUCTION

In the medical world, it is not always easy to explain what is happening without going far and deep into matter that can be hard to understand for people. In order to simplify and visualize these subjects, it can be worth looking at other, more well-known phenomena.

Cardiac excitation, which describes the electrical impulse making the heart muscles contract, is one of those things that can be hard to get a grip on. On top of that, when talking about normal and abnormal heart rhythms, having something to visualize these to patients or students might be beneficial to their understanding of what is happening.

As the electrical impulse moving over the heart is, in fact, a wave being moved by a certain wave propagation formula, using this main trait - wave propagation - makes it easy to find another, more widely known and easy-to-imagine event: wildfire.

In this paper, existing simulators will be looked into and compared with each other in order to determine what method would be the most effective way to make a real-time application. It will also go over how an actual wildfire behaves and research all of its properties. These elements will then be used to make the wildfire simulator, striking a balance between performance, its visualization purposes and the realistic side of a fire.

The application will need to be able to deal with terrain that is fully 3D, such as a sphere. It is also the goal to be able to run the simulation real-time, making sure there is no render process taking place. Using a 3D terrain will mean that some wildfire properties will lose their meaning. However, as this simulator is meant to show cardiac excitation in a more comprehensible way, having the simulation be too realistic could again make it hard to understand, which would contradict its goal.

LITERATURE STUDY

In the following sections, the properties of a real wildfire will be taken a look at. These properties, however often very complex in how they work and influence each other, are an important part of understanding how wildfires start, spread, and how intense they are.
After this, the paper will go over some of the existing simulators, diving into the methods they use to calculate the behavior of a wildfire. The last part will be about a less realistic, but very performant simulator, used in a game.

1. WILDFIRES: WHAT INFLUENCES THEM?

While a wildfire can appear to be something simple – “it burns anything it can use as fuel and travels in the direction of the wind” – the opposite is true. There are many factors playing a role in the start, spread, speed and lifetime of a fire.

THE FIRE TRIANGLE [1]

Fig. 2 The fire triangle: oxygen, heat and fuel [18]

To start a fire, you need three things: fuel, heat, and oxygen. During a hot, dry summer, plants and any loose material like branches and leaves form the perfect tinder. In the open air, oxygen is always present, and with that, the only thing still needed to start a wildfire is a spark.
While this spark can be caused by something like lightning, or even the heat of the sun burning down onto some dry branches, most of the time the wildfires are started by human carelessness. A discarded cigarette, campfire or fireworks are often the cause of a destructive fire.

WIND [2]

Wind affects wildfires in multiple ways.
The most obvious one is of course the pressure physically pushing flames and sparks in wind direction, possibly even causing new fires. The ignition of a new wildfire caused by embers from an already existing one is called spotting and can be extremely dangerous. The number of sparks and the distance they can travel depend on the type of fuel and the wind speed.
However, the wind does not only push the flames and sparks. It also pushes the heat in a certain direction, already pre-heating fuel that is not burning yet, drying it out and causing it to catch on fire quicker. The wind itself also causes evaporation of surface humidity, again making more fuel available to the fire at a more rapid rate.
Wind also influences one of the factors in the fire triangle, namely oxygen. As a fire burns, it uses up the oxygen in its direct environment. This oxygen gets refilled by wind, and the stronger the wind, the more oxygen is pushed into the fire, making it burn more intensely.

FLAMMAGENITUS CLOUDS

As wood burns, it releases a lot of moisture, which is then carried upwards with the heat of the wildfire. When the air above a fire is saturated enough, clouds known as “flammagenitus clouds” may form, as seen in Fig. 3.
Depending on the size and intensity of the wildfire, these clouds can take one of two routes.
In the best-case scenario, they condense so much they cause heavy rainfall, possibly extinguishing the

wildfire. [3], [4].

Fig. 3 A flammagenitus cloud. The dark portion is still smoke, the white, top part is the cloud [5].

However, if the fire is large enough, the flammagenitus instead evolves into a cumulonimbus flammagenitus, which can cause increased windspeeds, turbulence, and lightning, often without rainfall. This means that the already existing fire can increase in intensity, the risk of spotting can be higher, and lighting may cause entirely new fires. [4], [6]

SLOPE

Slope can affect the spread rate of a fire tremendously. As a fire goes uphill, heat rises and dries out the fuel in front of the fire. The steeper the slope, the more heat can reach this fuel, effectively making the fire spread faster as fuel has been pre-dried and heated up [1].
On top of that, slopes often cause drafts: local winds that are determined by the slope and temperature of the surface. During the day, the surface warms up and the warm air rises up-slope. At night, the surface cools and the opposite happens, where down-slope drafts occurs. These drafts cause fires to go uphill during the day, together with the effect of pre-heating intensifying the fire [7].

FUEL [8], [9]

Type, size, moisture and arrangement of fuel all play a role in a wildfire. Some trees might burn faster or produce more embers, others might contain more moisture and therefore slow down a fire. Thin branches will catch on fire more quickly than a log, and dry pine needles will ignite faster than fresh ones would. A patchy grassland will slow down or even burn out a fire, while dense woodlands can turn into a raging fire.

CROWN, SURFACE AND GROUND FIRE [10]

Wildfire can roughly be divided in three categories, namely crown fire, surface fire, and ground fire. Crown fires burn the canopy, effectively setting whole trees ablaze. This type of fire is the most intense and difficult to obtain. It often spreads more rapidly than surface fire, as it is more exposed to wind, but also needs the most fuel.
Surface fires burn fuel on the ground. This can be shrubs, branches, leaves, pine needles… Often , these fires are the easiest to contain and don't spread that fast. They are however influenced the most by slope, and can suddenly increase in spread rate when going uphill.
Ground fires, on the other hand, are fires that get underground and burn roots, organic material, coal, and other buried flammable matter. These fires burn the slowest and longest, and are difficult to contain.

2. EXISTING WILDFIRE SIMULATORS

Wildfire simulators are used in the prediction, prevention and containment of wildfires. They can take in input with info about weather, fuel, and topography, and use this info to simulate how a fire would behave under these circumstances. This data can then be used to identify high-risk areas, predict the further path of a fire, and single out locations that are suitable for placing.
Below, a few of the common simulators are looked into.

FARSITE [11], [12]

FARSITE, used by the United States Department of Agriculture (USDA), is a wildfire simulator that uses existing models for surface fire, crown fire and spotting.
This simulator uses Huygens' principle to predict the shape of the fire front. This means using an elliptical shape as the base, and on every vertex of this ellipse, a wavelet is placed as illustrated in Fig. 4. This wavelet is then influenced by factors like wind and slope, after which the final shape of the fire front can be computed. The ignition point forms the base of this elliptical shape, and multiple ellipses can merge together into one fire.

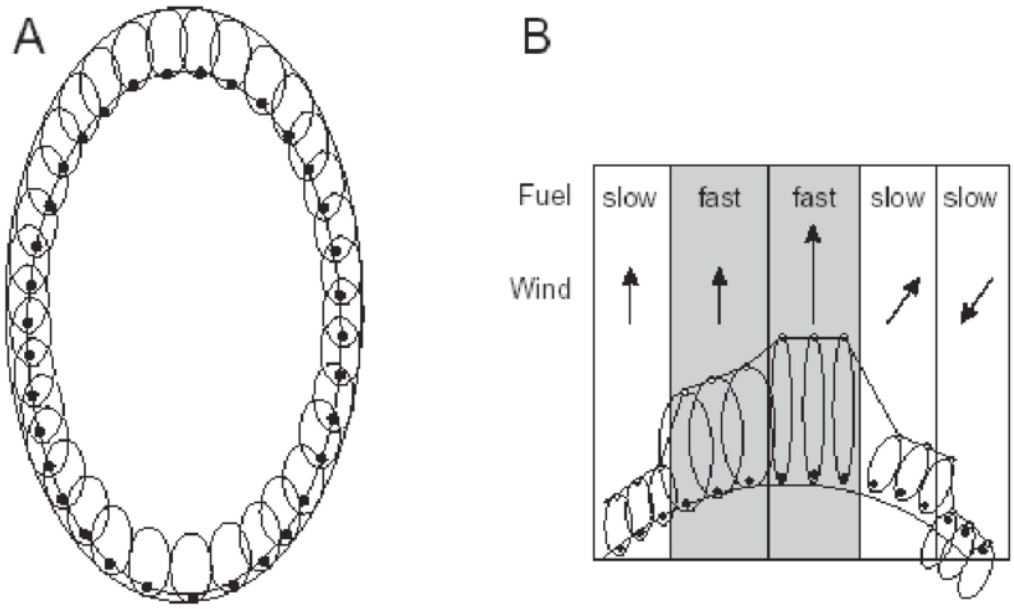


Fig. 4 An ellipse with wavelets on each vertex. The wavelets can be individually influenced [19].

Due to the amount of input (topography, wind, weather and fuel) and the period of time for which FARSITE computes, a single simulation can take up to a few hours.

PROMETHEUS [9]

Prometheus is used in Canada by the Canadian Forest Service. Just like FARSITE, it uses Huygens' principle to compute the simulation. The main difference lies in the input, as e.g. fuel models will differ between Prometheus and FARSITE.
Again, this simulator, while relatively accurate, will take up to several hours to compute a full simulation.

CELL2FIRE [13]

Unlike FARSITE and Prometheus, Cell2Fire uses a grid-based approach for its simulations. It divides a landscape into cells, assigns information to these cells, and then, using the center of one cell as ignition point, starts calculating ellipses. Note that, while it does use elliptical shapes, it does not use Huygens' principle, which uses wavelets on these ellipses to calculate the fire front shape. When the ellipse reaches the center of another cell, this cell itself becomes an ignition point, and the calculations from this ignition point will now use the data from this cell. The ellipses themselves are formed by different Rate of Spread (ROS): head (HROS), back (BROS) and flank (FROS). This way, data like wind and slope are incorporated.
Cell2Fire can compute larger scale simulations then Prometheus and FARSITE, as its calculations are generally simpler. It is also faster than the simulators that use Huygens' principle. Test results indicate that the accuracy from Cell2Fire is pretty high and therefore useful in real-life applications.

WAVE PROPAGATION

Wave propagation in itself simply is "the way in which a wave travels". This can be any kind of wave, like seismic waves, sound, or the waves that are created by dropping a stone in a pool of water. Waves interact with materials and obstacles in certain ways, and when overlapping, can influence each other's strength [14].
Of course, not all of this directly applies to wildfires, but the way a fire front moves can still be represented by wave propagation. Huygens' principle relies on this, as the wavelets on the base ellipse are shaped by factors like weather and topography to then form the fire, or wave front [9], [11].
Cell2Fire also uses it, although in a simplified manner, by calculating the HROS, BROS and FROS and applying this to every elliptical shape [13].

3. WILDFIRE IN GAMES [15]

Sometimes, other systems also require wildfire simulations. Games can use them as gameplay element. One of these games is Far Cry 2, where you can set fields and objects ablaze by for example throwing Molotov cocktails.
Although the wildfire simulation in the Far Cry 2 game is on a much smaller scale, the simple fact that it needs to run alongside a full blown AAA game makes it an interesting use case.
The simulator is entirely cell-based and, true to a game, uses damage, hitpoints, and energy to decide whether a cell catches on fire, burns, or is burned out. These factors also decide the propagation of the fire throughout the grid: a cell with high hitpoints will catch on fire much more slowly than a cell with low hitpoints, and therefore slow down the fire. On the other hand, a cell that has a lot of energy (fuel) will burn long and will therefore have a chance to do much more damage to neighboring cells.

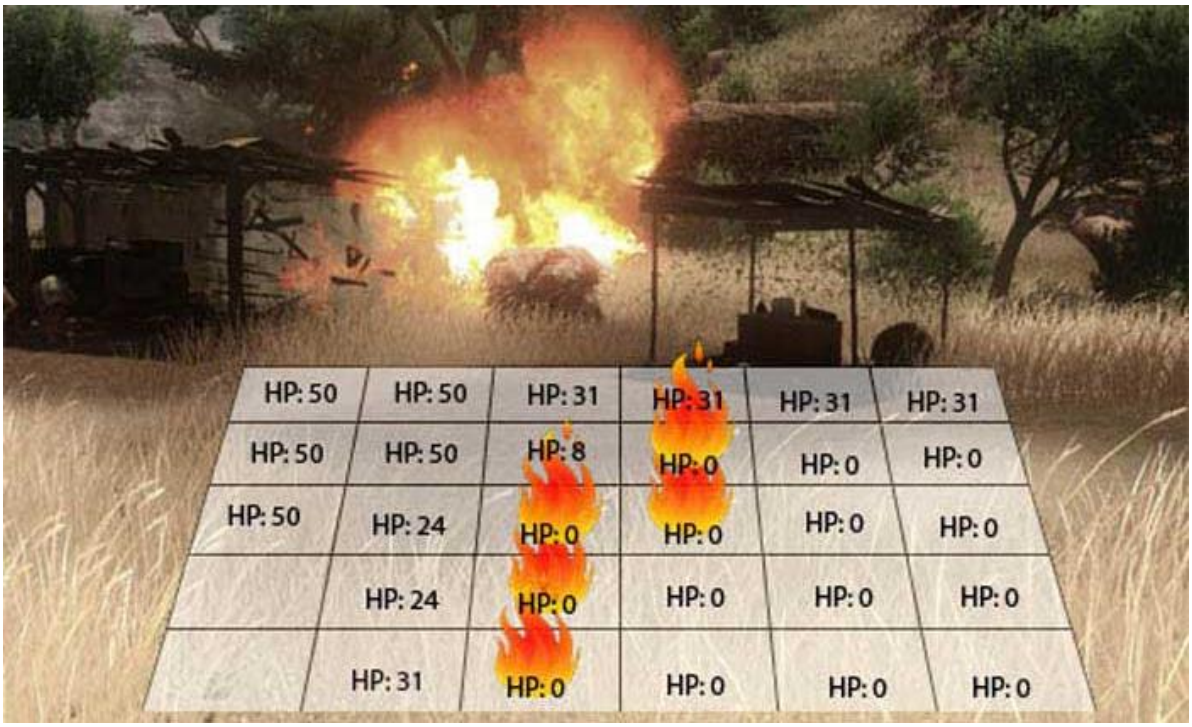


Fig. 5 Demonstration of the in-game fire grid and its workings [14].

Wind is added by vectorizing it and calculating a dot product with the direction from cell to cell and the wind direction. Cells that are downwind from a burning cell will receive more damage than cells on the other side of the fire.
In the case of the Far Cry 2 game, these fire grids are fairly small, meant to encapsulate patches of grass and objects no larger than a house, however, the system is easily scalable to full terrains, as the need of running gameplay would fall away in a pure simulator.

CASE STUDY

1. INTRODUCTION

Usually, the methods used in currently existing, realistic wildfire simulators are slow, easily running up to a few hours to calculate their results. As such, using their math-heavy techniques was not possible, as the goal is to make an application that shows fire propagation in real time. Because of this, the Far Cry 2 simulator was far more interesting, and is – more or less – what ended up being used.
In FC2, the grids are a lot more detailed, causing fire to spread over objects like chairs and such, while the aim of this application is to simulate a wildfire on a terrain. However, as the technique can easily be scaled up, this did not pose a lot of problems.

As for the application itself, some UI would need to be added in order to be able to run and use the software without having to return to the codebase every single time. This also meant that a few things would need to be implemented purely for this useability.

2. SPECS AND SOFTWARE

The simulator has been made and tested on a desktop with the following specs:

CPU	Intel(R) Core(TM) i7-7700 CPU 3.60 GHz
RAM	16 GB Corsair VENGEANCE DDR4 2133 MHz (2 x 8 GB)
GPU	3 GB NVIDIA GEFORCE GTX 1060

Furthermore, the following software has been used: Unreal Engine 4.27.2 , Visual Studio 2019, and Blender.

3. SIMULATION

Throughout the making of the application, several steps were taken, and oftentimes retaken to fix or optimize a few things. In the following parts, every step will be explained in a more or less chronological order.

THE GRID

Two grids are used: one that is more directed towards the user, and one that would actually do the fire propagation. One cell in the large grid, the parent grid, holds the smaller, second grid or child grid. Every parent cell decides properties of all of its child cells. This double grid was chosen to make setting

properties easier, as the total amount of child cells when counting all child grids together easily goes over a thousand. This setup could also be used as a form of spatial division, to lower the calculation cost. However, Unreal Engine already provides options to choose whether or not a cell should tick and as such, every child cell is in charge of its own ticking behavior.

THE FIRST GRID GENERATION ALGORITHM

The first algorithm that was created in order to correctly generate the grid was a very straightforward, naïve solution that is the easiest to understand, as it is very clearly divided in all its subparts.

Firstly, the maximum bounds of the objects that need to be enveloped are calculated. Unreal Engine already gives you a similar function per actor, so all that needed to happen was to get all of the bounds and create one that held all of those. This is the total space that needs to be filled with a grid. Cells that would fall entirely outside should not be used in the simulator and would only eat fps and memory.

After this, the size of the parent cells (given by the user) is taken, the number of cells for width, height and depth are calculated and the grid is put in place. In terms of Unreal Engine C++, a UBoxComponent was used to visualize the cells. This also makes it easy to do overlaps, and certain info such as origin and half extents is already kept.
However, even taking into account the maximum bounds of the terrain object, there are still a lot of cells that are redundant, and can even mess with the wave propagation. For example, there should not be any fire travelling on the inside of a mesh.

To fix this, once the parent grid is generated, an overlap test against the terrain actors is done. If a cell overlaps, it means it actually *needs* to be able to run the simulation on the terrain. Otherwise, the cell is removed from the grid altogether. After this step, a result as illustrated in fig. 6 is achieved.

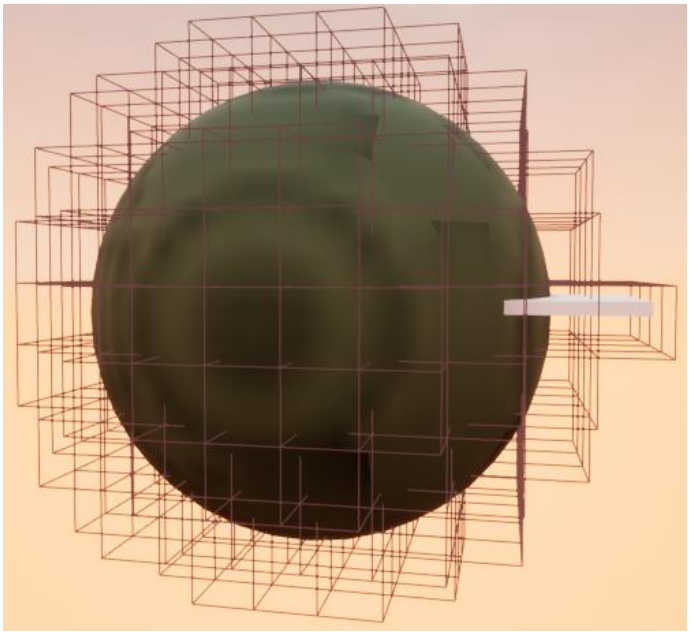


Fig. 6 Visualization of how a grid envelops multiple actors.

The only cells remaining are the ones that are still overlapping the terrain, in this case a large sphere and a thin beam. The grid is hollow as well, so fire cannot travel via cells on the inside.
Some notes on how the overlap in UE4 works: this kind of overlap, where it does not see a shape completely inside another shape as overlapping, will only work when using complex collision. This means UE will use the geometry of the mesh itself to check for collision, instead of a simplified encapsulation of it. This makes it so the polygons are tested for overlap, and if they do not register overlap (for example when the shape is completely inside), it will not trigger anything. If the mesh were to use simple collision, this would not work. On top of that, as the cells are spawned in place and eliminated in the same frame, it is necessary to call the update on the overlap behind each cell manually. For the update overlap to work properly, after startup of the application, a small time delay should be used before generating the grid.

When all of this is done, the child grid is generated in much the same way as the parent grid. The only difference is that, with these child cells, the number per row is given, and the size is calculated from there. All of the cells are kept cubical in shape.

The problem with this algorithm is that it takes quite a bit of time. Even worse, Unreal would often freeze on it, or internally something would go wrong and the grid would be generated and immediately deleted, as overlap updates seemed to be ignored. However, it was a good start to understand how to generate the grid in a way that would minimize the calculations of the fire.

THE SECOND GRID GENERATION ALGORITHM

The main cause of the first algorithm being so time consuming, came down to memory management. When creating the grid, lots of cells are initialized, often to be destroyed immediately. Not only was there a huge amount of initialization and destruction, the array that was keeping the cells was also constantly changing in size, as items at any index were removed during the elimination of the cells. In other words, if the way the cells were spawned and deleted could be changed, a lot of time could be gained.

The second algorithm, also the one that is currently used in the application, swaps the order of steps around. It *first* checks for overlap, with one of UE4's overlap functions. It does not have to initialize an actor on the heap, nor does it need to delete this memory afterwards. Only when there is overlap, a cell is spawned. The location at which this cell is spawned and how that location is calculated stays the same as in the first algorithm.
The child grid uses the same principle; first checking for overlap before deciding whether to spawn or not.

This way of working also means that the arrays keeping pointers to the cells do not have to change size anymore after the grid has been spawned. There are no remove operation necessary, and the time it takes for the grid to be generated is cut by a lot. The small time delay is still necessary, but it rarely freezes or ignores the overlaps. It is possible that the freezes occur only because of the overhead when working and testing in editor.

The end result looks the exact same as the end result of the first algorithm.

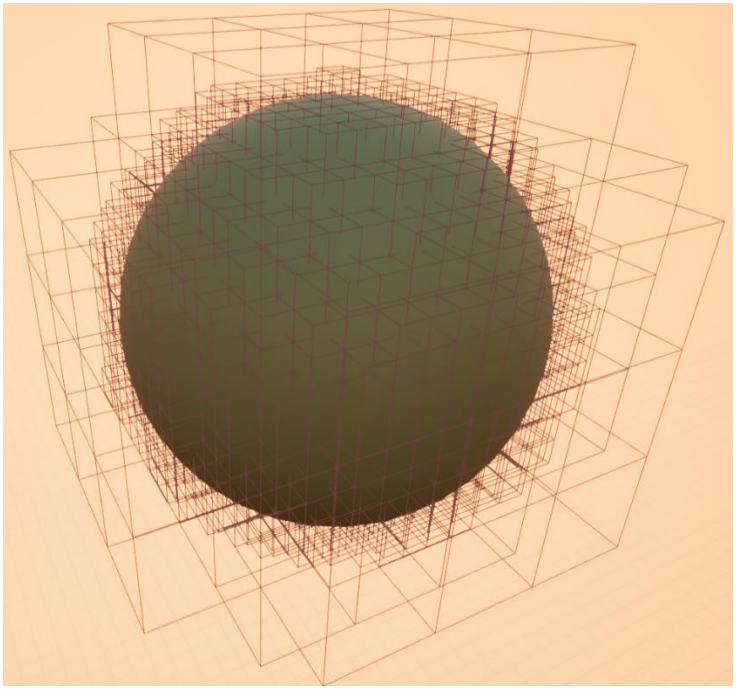


Fig. 7 Parent and child grid, adapted to underlying geometry.

In pseudocode, the algorithm would look something like this:

```
//PARENT:
for (all parent cells)
    destroy child grid and destroy the cell
reset array

//PARENT:
figure out the origin and half extents of all terrain actors together
use this to find how many columns, rows and depth is needed

//CHILD:
figure out the size of the cell

for (every row)
   for (every column)
      for (every depth)
         do a box overlap with terrain.
         if there is overlap, spawn cell
         set cell stats
         change Y location
      reset Y
      change X location
```

```
reset X
change Z location
```

CELL DATA: NEIGHBOURS

When the fire is starting to spread, iterating over all (child) cells and looking which ones are valid targets to do fire damage to would be a waste of resources. In order to avoid this, every child cell keeps an array with pointers to its neighbours, and when it is set on fire, it will only loop over those.

At first, when using the first grid generation algorithm, the cells would figure out their neighbours by using math on their index in the grid. While it was possible to find the neighbouring cells in its simplest form, several issues popped up with this approach.
It was nearly impossible to properly detect cells that would "go over the edge", popping up on the other side of the grid. Beside this, there was also the problem of the child grid essentially being several grids put next to each other. As the elimination process would also happen, the arrays keeping neighbours would then need to be cleaned, removing any cells that were marked for deletion by the garbage collector.
In short, there was no proper way to directly figure out the neighbours by just using indexing.

Using Unreal Engine's functionality, a box overlap is done with a box that is slightly bigger than the cell doing the overlap. This means it will overlap every actor that is next to it, effectively collecting all of the neighbours. The returned array of overlapping actors is then filtered for child cells, and all the found child cells are stored as neighbours. As with the grid generation algorithm, no removal of items is now needed.

CELL BEHAVIOR

Both parent cells and child cells have their specific role in the application. Parent cells are focused on giving the user a way of easily setting properties on a bunch of child cells at once, while the child cells are the ones who do the actual fire propagation.

PARENT CELLS

Parent cells provide a gateway to the properties of its child grid. Whenever the user makes a change to the properties of the parent cell, all of these changes are propagated to the child grid. These properties include fuel, fire resistance, wind velocity, tick rate and number of child cells per row.

These cells also give the user the option to change the size of the parent cell and holds the code to spawn, reset, and empty the child grid.

When the user chooses an ignition point, which is at child cell level, the double grid is effectively used as spatial partitioning. First, it finds out what parent cell the clicked location is in, and then it looks in the child grid of this cell to figure out the exact child cell.

CHILD CELLS

This is where the core of the simulation is. Upon spawning, the cell is not ticking and is in a clean, reset state. The tree, used for visualization, is fully visible, fuel and resistance are at their maximum, and no cell is an ignition point.

When the user chooses an ignition point and starts the simulation, the picked cell will start ticking, the fire VFX is activated and the crown of the tree is set invisible. Using a tick rate, chosen by the user or default 0.5s, it will do damage to its neighbours. This damage is calculated with a dot product using the direction of the fire and the wind velocity. The direction from damaging cell to damaged cell is normalized, while the wind velocity is not; this is to keep the strength of the wind a factor as well. If the result of this calculation is less than 1, the damage value is set to 1. This way, the fire will always damage its neighbouring cells, but will spread much slower than it would when going in the direction of the wind.

The cell which got damaged will subtract the damage value from its fire resistance. Once this fire resistance becomes 0 or negative, the cell becomes an ignition point itself, ticking and switching its state to burning.

Fig. 8 Fire propagating over the terrain. The blank terrain has been burned down.

An issue that appeared with smaller values for fuel and resistance when relying purely on the built-in tick rate that UE4 gives the user access to, was that a chain reaction could be triggered which would set all of the cells on fire within a few frames. This happens because the tick interval is only used after the first tick of an object, meaning that if the application was running at 144 fps, a cell catching on fire would damage its neighbours after only 0.006 seconds. If fuel and fire resistance are low enough, this meant that the damaged cells would also immediately catch on fire, starting their tick and damaging their neighbours after another 0.006 seconds, until the whole terrain caught on fire.
This issue was resolved by starting a timer instead of immediately enabling the tick. This timer takes care of the first “tick” interval, effectively halting any chain reaction that would have happened before.

While a cell is doing damage, every time it ticks, it will subtract 1 from its fuel. Once the pool of fuel is empty, the cell is burned out, the VFX is deactivated and the trunk of the tree also disappears. A burned out cell cannot catch on fire again until it is reset.

VISUALIZATION: FIRE VFX

To properly visualize a fire, some kind of VFX is needed. However, VFX are often very taxing on fps, and the child cell count could easily go beyond a thousand, depending on how precise the grid is. A lightweight fire VFX was needed. Using a tutorial as a base [16] and UE's shader complexity view to check the cost of the VFX, a flexible yet cheap fire was created.

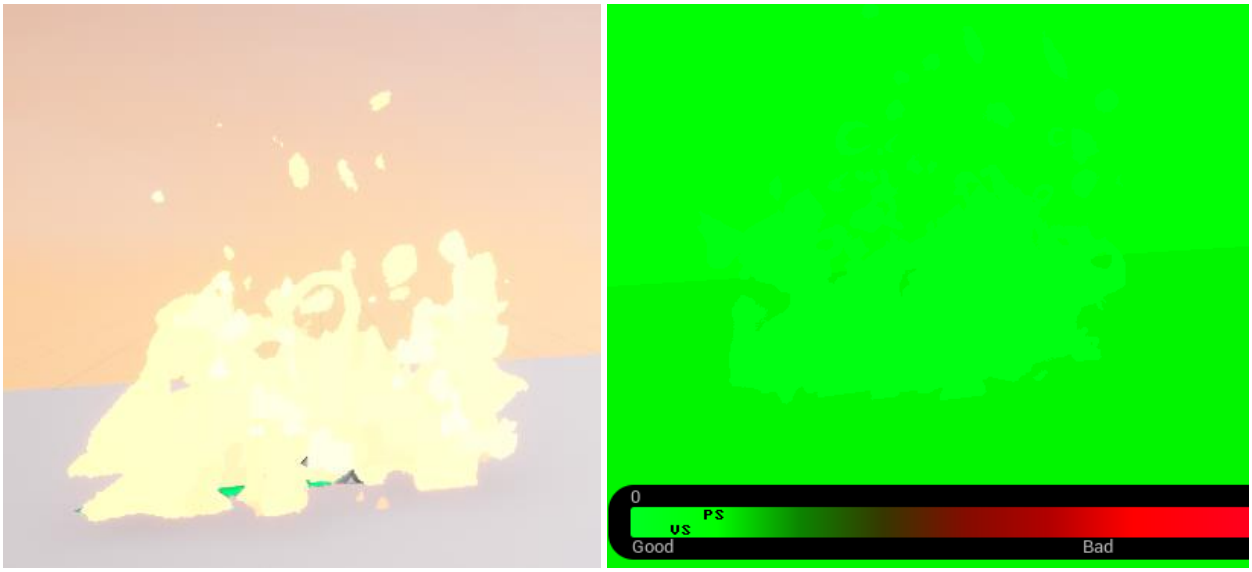


Fig. 9 The Niagara fire VFX and how it looks when visualizing shader complexity.

INTERSECTION OF CELL WITH TERRAIN

One problem remained: the fire as well as the tree mesh would always stand upright according to the axi of UE. However, working with a full 360 terrain, positive world z is not necessarily up.

Somehow, an intersection point needed to be calculated. The cells are always cubic in shape, but the terrain could be anything. The algorithm needed to be able to deal with this.

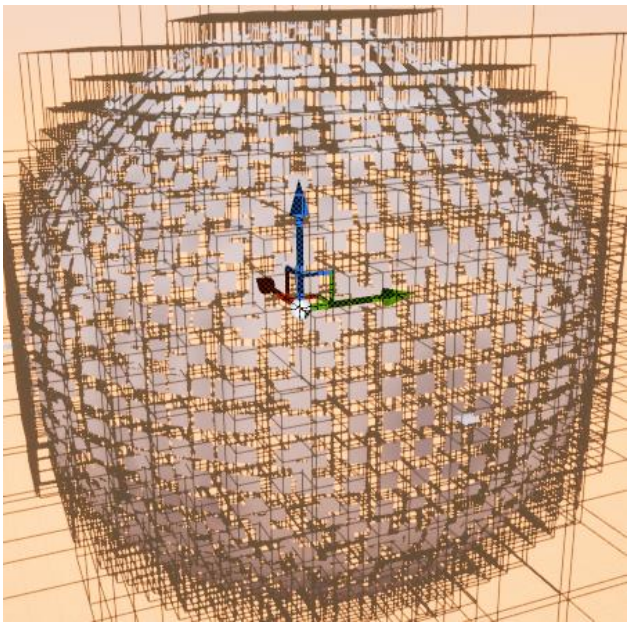
LINETRACING CENTER TO CENTER

The first, most obvious solution was to do a few line traces. Line traces can catch any geometry when checking against complex collision, which would be ideal to find a decent location for tree and fire. The line traces used the center of each side of the cube, going from one side to the other. However, this method left out too many cells that were overlapping with only their corners and did not trigger any blocking hits on the line traces.

LINETRACING EDGES

There might be a lot of cells that only overlap with a corner, there aren't many that only overlap with a part of a side. Therefore, while line tracing edges meant many more traces, it should catch blocking hits in almost all cells.

In order to easily loop over all the edges, a simplified vertex and index buffer was made. The index buffer doesn't hold polygons, but edges instead.
Every edge is then line traced on a channel specifically for the terrain, so no other hits could be detected, and all blocking locations and normals are kept in a temporary array.



The problem with this approach is that traces from inside the terrain mesh to the outside would not register any hit, and a lot of data was lost this way. Using double sided geometry as an option on the static mesh in UE was also not possible. To properly orient the fire and tree, a normal vector was needed as well, and the double sided geometry meant that the hit normals could point both inwards and outwards.
In the end, every edge is being traced twice, unless a blocking hit was found on the first trace.

After all hit locations and normals have been collected, they are averaged out to obtain the final intersection point and rotation. This data is then used to place the fire VFX and the tree mesh.

Fig. 10 Visualization of the intersection algorithm. The terrain has been set to invisible, the planes show where the intersections points are and how they are oriented.

This algorithm, in pseudocode, looks like this:

```
vertex array[8] filled in with all vertex locations
index array[12] filled in with every pair of vertices (indexing in vertex
array) that makes an edge

for (every edge stored in index array)
    linetrace edge in one direction
    if hit, save impact location and impact normal and continue

    if there was no hit, linetrace in opposite direction
    if hit, save impact location and impact normal

average out the collected impact locations
average out the collected impact normals
```

4. OBSERVER

The observer is, in the application, the instance that "watches" the fire simulation. It is merely a camera mounted on a spring arm (USpringArmComponent in UE) that provides the functionality to make movement possible.

Apart from that, it is also the interface between the UI and the properties of the grids. Any change in the UI, like clicking a button or inserting a number in an input field, is propagated to the observer, which then processes the new data.

It also enables the user to select cells, using the cursor location in screen space, converting this to world space, and doing a line trace from camera to this world space cursor location. With this, it controls both the selection of an ignition (child) cell and the selection of a parent cell.

5. UI

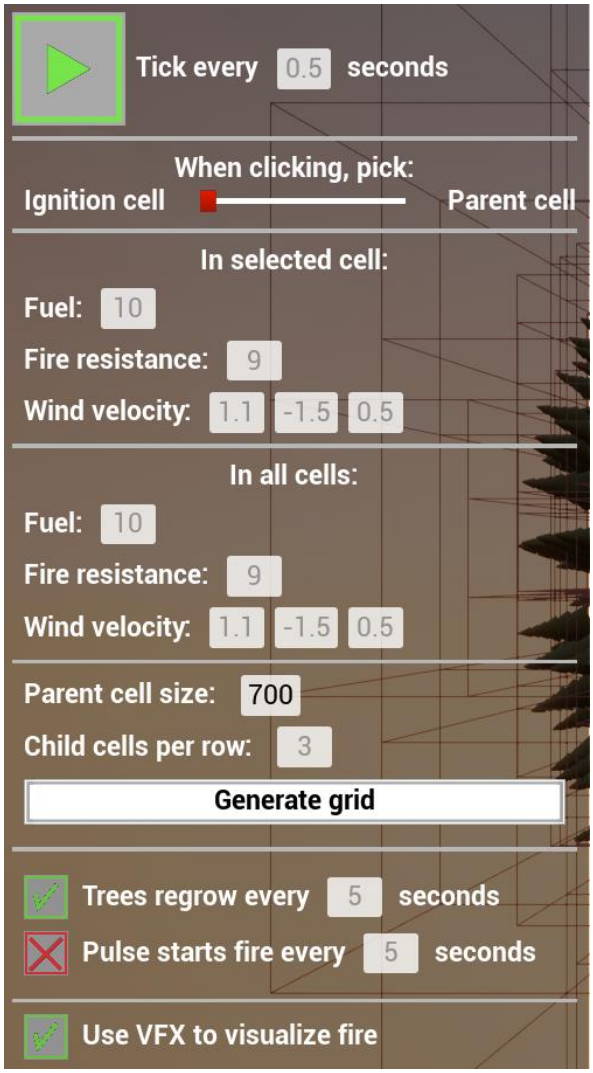


Fig. 11 The UI when all options are visible.

The UI, as seen in Fig.11, enables the user to quickly make and test different scenarios. It provides different options to set the properties of the grid and lets the user start and stop the simulation while the application itself can keep running.

START, STOP AND TICK RATE

When starting the application, the simulation will not automatically start. Instead, it will wait for the user to click the play button. Once running, the button changes to a stop button.
The tick rate is default 0.5 seconds, but can be set to any non-negative value by the user.

SELECTION

The slider decides what will be selected on a click on the terrain. When picking an ignition cell, a child cell will be marked as ignition point, and a simple VFX will appear in the right spot to visualize this.
When picking a parent cell, the cell will be highlighted and the properties of this cell's child grid can be set.

PROPERTIES

The values in the block "In selected cell" will only affect the selected parent cell. If no cell has been selected, nothing will happen.

When inputting values in the second block, all cells will be given this value. If cells had been selected previously, their values will also change.

GRID SIZE

As mentioned in the grid generation part, the user can choose the size of the parent cell, but not the number per row, column and depth. For the child grid, the user can choose how many there are on one row. As the parent cell is cubical, this is all the necessary data. When values have been inputted, the "Generate grid" button appears. If clicked, the grid will be entirely regenerated, using the new values.

REGROW AND PULSE

The user can also choose to have the trees regrow after a few seconds. If the trees can regrow, the second option appears and the simulation can reignite the fire every x seconds. With this, the simulation can keep running instead of ending as soon as all cells have been burned out.

USE VFX

As VFX are still something that has a significant impact on fps, no matter how cheap the effect itself is, an option to turn them off has been included. The trees will still change, but there won't be any fire visible. This can be used to counter the weight of very detailed grids, or to make it run on lighter machines.

EXPERIMENTS & RESULTS

Several tests have been conducted in order to figure out whether or not the application is capable of running the simulation in real-time. After the performance tests, multiple cases have been tested to see if the algorithms for grid generation and intersection point finding hold up when fed different terrain meshes. Below, all of the results can be found.

FPS

The main goal of the application is to be able to run the simulation in real-time, without experiencing lag or needing to wait for a render process. In order to test this, a few cases have been used to collect performance data. All testing has been done in-editor, which runs at a default 120 fps, and on the computer setup that was previously mentioned. The both editor game screen and build screen have been put to full screen.

STRESS TESTING VFX

VFX are usually pretty heavy, even when they are made as lightweight as possible. The fire VFX in the case of the wildfire simulator is very light, but often needs to run several hundreds of instances at once, with a maximum of over a thousand at the same time. In order to see the effect of the VFX on the fps, stress testing was performed by putting the whole terrain on fire at once. The cells holding the fire are not ticking and thus not doing any simulation, the VFX has simply been turned on permanently. Table 1 and Chart 1 visualize the collected data.

	parent cell size	child cell number	total number of child cells	fps
1	1000	3	674	57
2	1000	4	1165	35
3	1000	5	1898	22
4	750	3	1250	33
5	750	4	2123	20
6	750	5	3354	13

Table 1: Number of child cells and their influence on fps.

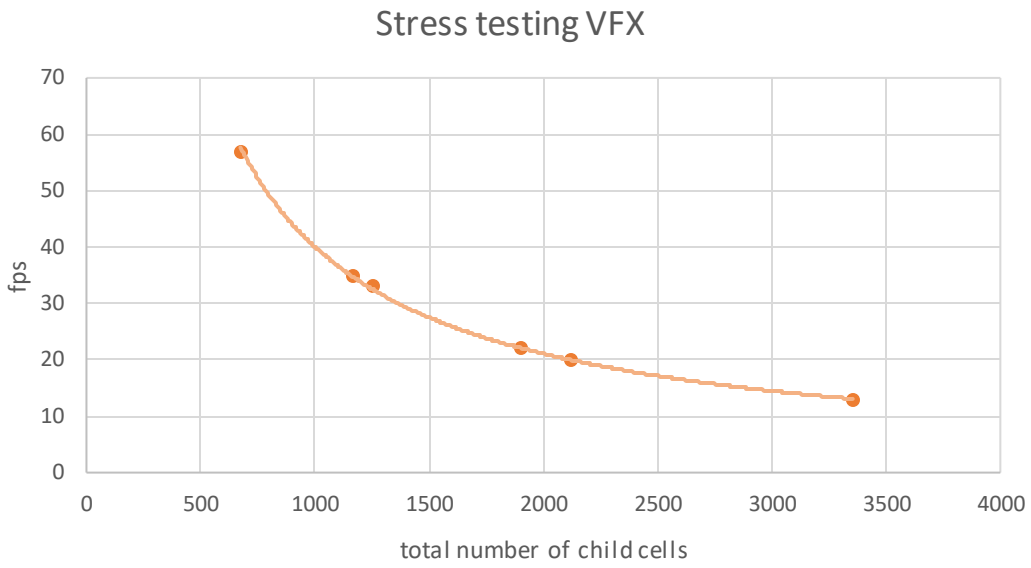


Chart 1: Number of child cells and their influence on fps, visualized on a chart.

SIMULATION: ONE CYCLE

Usually, the terrain will not be completely on fire during a simulation. There will be patches of fire, places where the fire did not yet reach, and places that have been burned down.

The following testcases use the same values for parent size and child number as the previous batch. Fuel has been set to 7, fire resistance to 6, and tick rate to 0.3. With these values, the simulation was run twice – once with VFX and once without – and an average fps was taken.
Again, the results have been visualized in a table (Table 2) and a chart (Chart 2).

	fps (VFX)	fps (no VFX)
1	108	120
2	73	105
3	53	76
4	68	97
5	46	65
6	34	47

Table 2: The influence of VFX on the fps in a one cycle simulation.

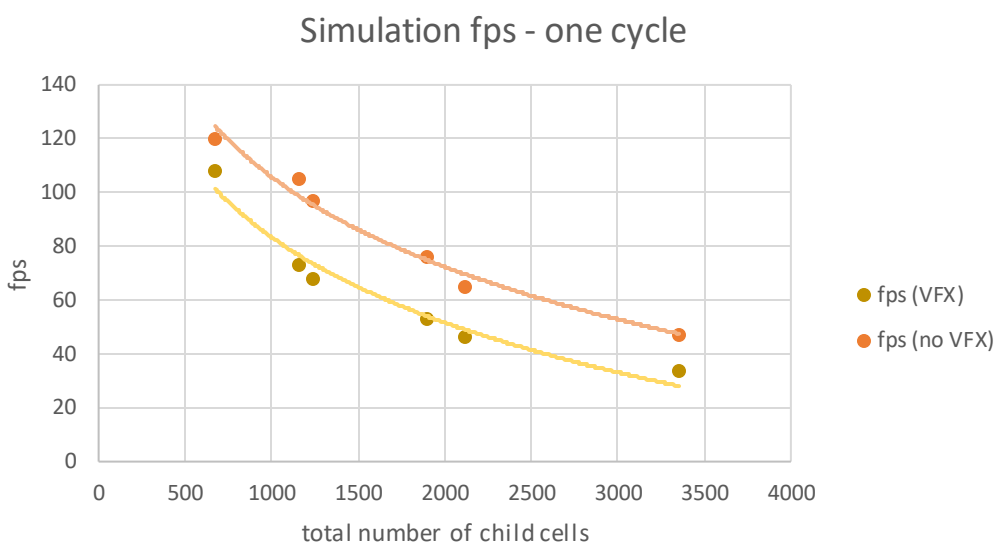


Chart 2: The influence of VFX on fps in one cycle, visualized in a chart.

SIMULATION: INCLUDING REGROWTH AND PULSE

When using the options to let trees regrow and the ignition point pulse, there isn't necessarily a large difference in how many effects and cells are active at the peak of fire coverage during simulation. However, when being able to regrow, every cell makes a new timer (UE4's timer functionality) which then ticks and calls a certain function – in this case a reset on the cell. This means that, even if the cell is not ticking, it still causes something else to tick, and a different fps is possible.

All values have been kept the same. Trees regrow 5 seconds after being burned out, and the ignition cell triggers a fire every 6 seconds. Results have been noted in Table 3 and visualized in Chart 3.

	fps (VFX)	fps (no VFX)
1	109	120
2	75	118
3	55	87
4	74	112
5	49	78
6	33	56

Table 3: Influence of VFX on the fps when cells are rest during simulation.

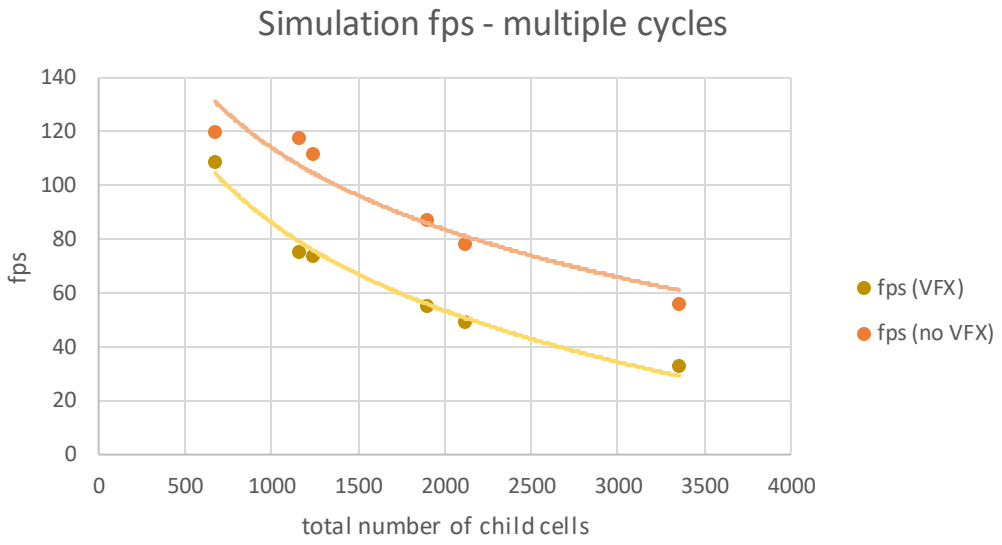


Chart 3: Influence of VFX on fps during regrowth and pulse simulation, visualized in a chart.

PERFORMANCE IN THE BUILD

Packaging the simulator means that, even if there is no overhead anymore, the quality of the visuals goes up, weighing on the fps more. The trees, which are spawned in at runtime and are thus dynamically lit, take up quite a bit more fps, noticeable when running the simulator without VFX. The starting fps (only full trees, no empty space and/or burning trunks) tends to give the lowest fps, and as trees burn and disappear, the fps goes up.
This time, the fps is not capped by the editor, so it can go beyond 120 fps. The same values have been used as in editor, and the fps over one cycle has been taken. The direct results have been noted down in Table 4, while Chart 4 and Chart 5 compare the data from the build with the data from the editor, both from one cycle.

	fps (VFX)	fps (no VFX)
1	127	182
2	93	134
3	58	92
4	86	126
5	55	77
6	36	49

Table 4: Running build with and without VFX, one cycle.

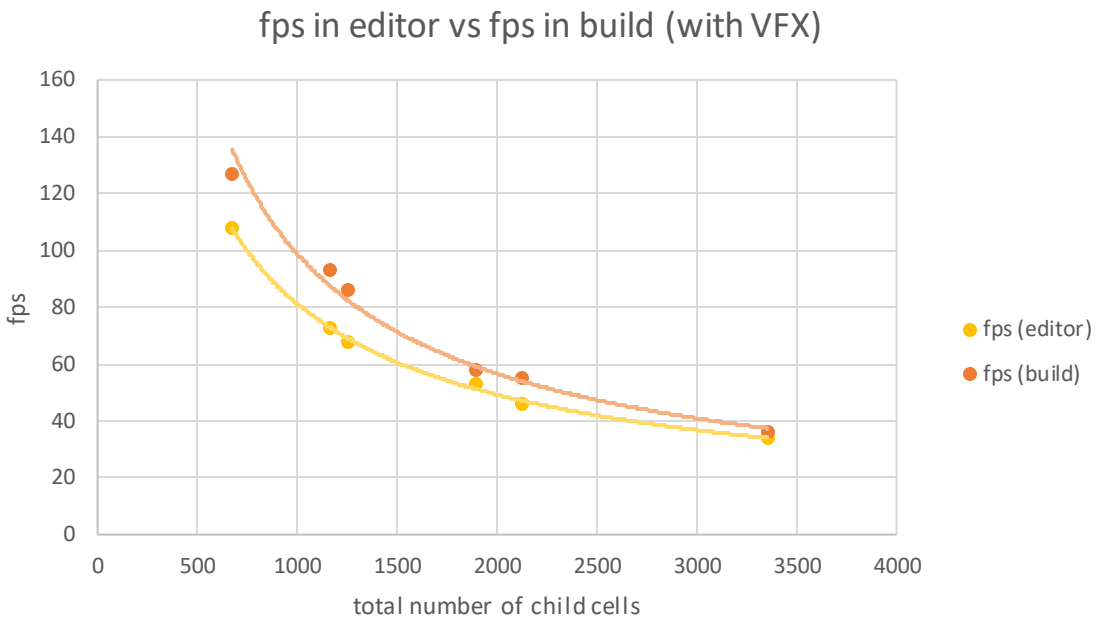


Chart 4: Comparison of the fps in editor and in build, running one cycle with VFX.

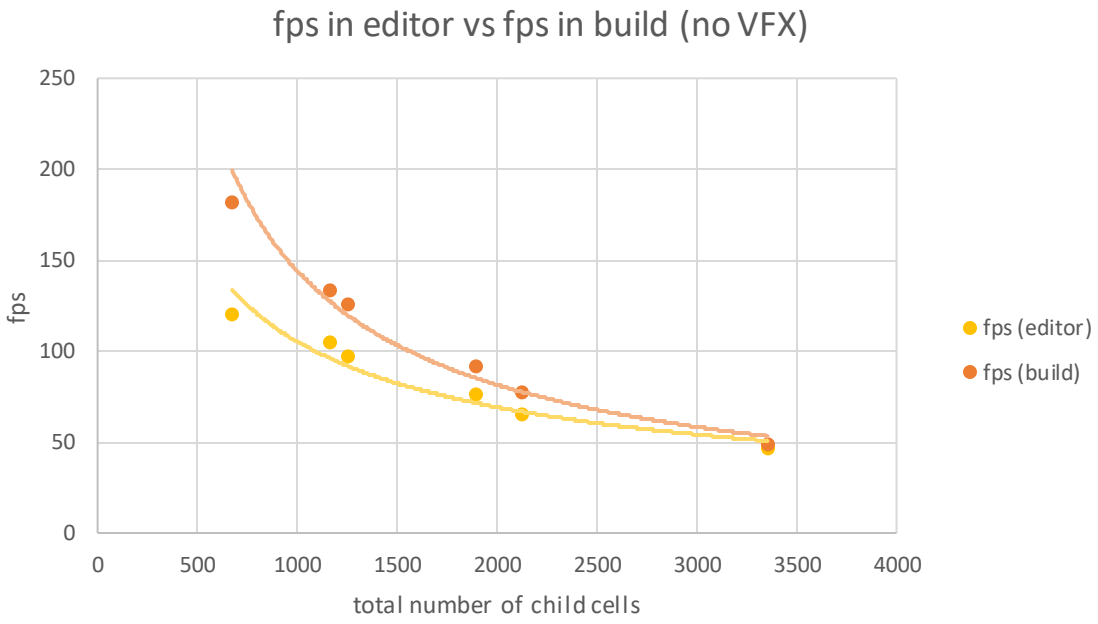


Chart 5: Comparison of the fps in editor and in build, running one cycle without VFX.

GRID GENERATION AND INTERSECTION POINTS

In order to properly deal with a full 3D terrain, the grid generation and intersection point algorithms need to be able to work with pretty much any mesh that is given to them. This means that some testing with models other then that of a sphere had to be done. As the intersection point algorithm is partially

dependent on the grid generation – it uses the child cells to do line tracing -, the two algorithms are tested at the same time.

SINGLE TERRAIN MESH

The application has been made using a basic sphere as terrain. However, the algorithms are built in such a way that they can be used on other shapes as well. To test this, Blender's monkey head and a model of a heart [17] have been plugged into the app.

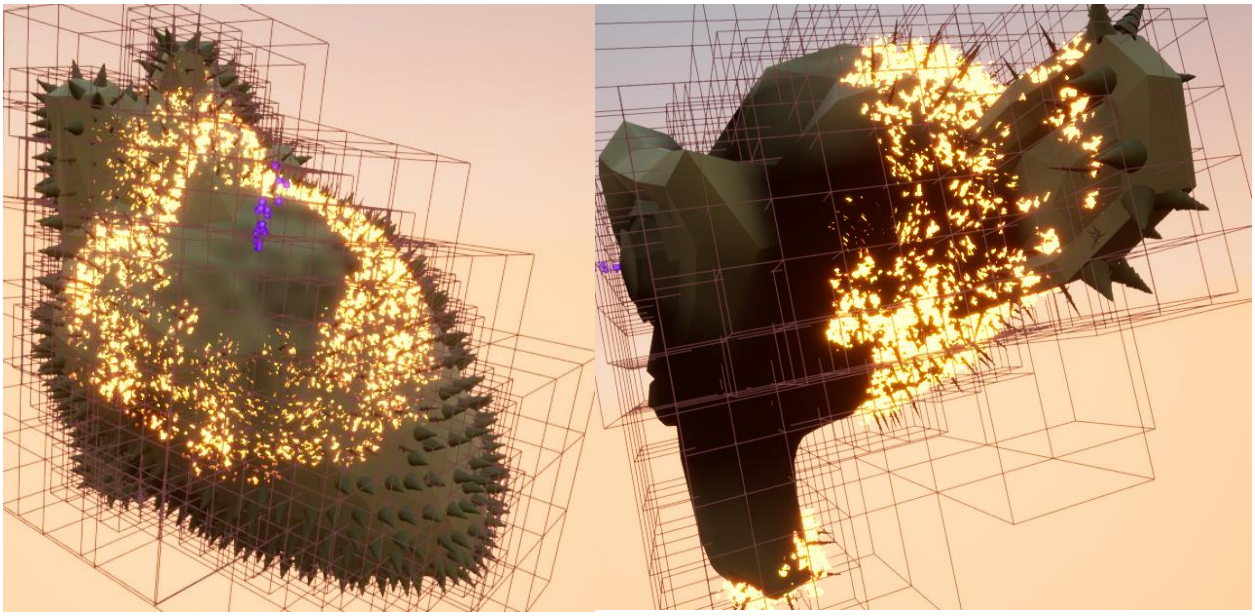


Fig. 12 Simulations running on other geometry (heart and monkey head).

When geometry has convex shapes or has a lot of ridges and bumps within one child cell, trees tend to clip through the terrain or appear floating. However, most of them are fine, being located in a good spot and pointing the right way. As visible in Fig. 12, grid generation and fire propagation work as intended.

MULTIPLE TERRAIN MESHES

Again using the monkey head and heart meshes, the application was tested putting two meshes next to one another and selecting them both as terrain to be used in the grid generation.

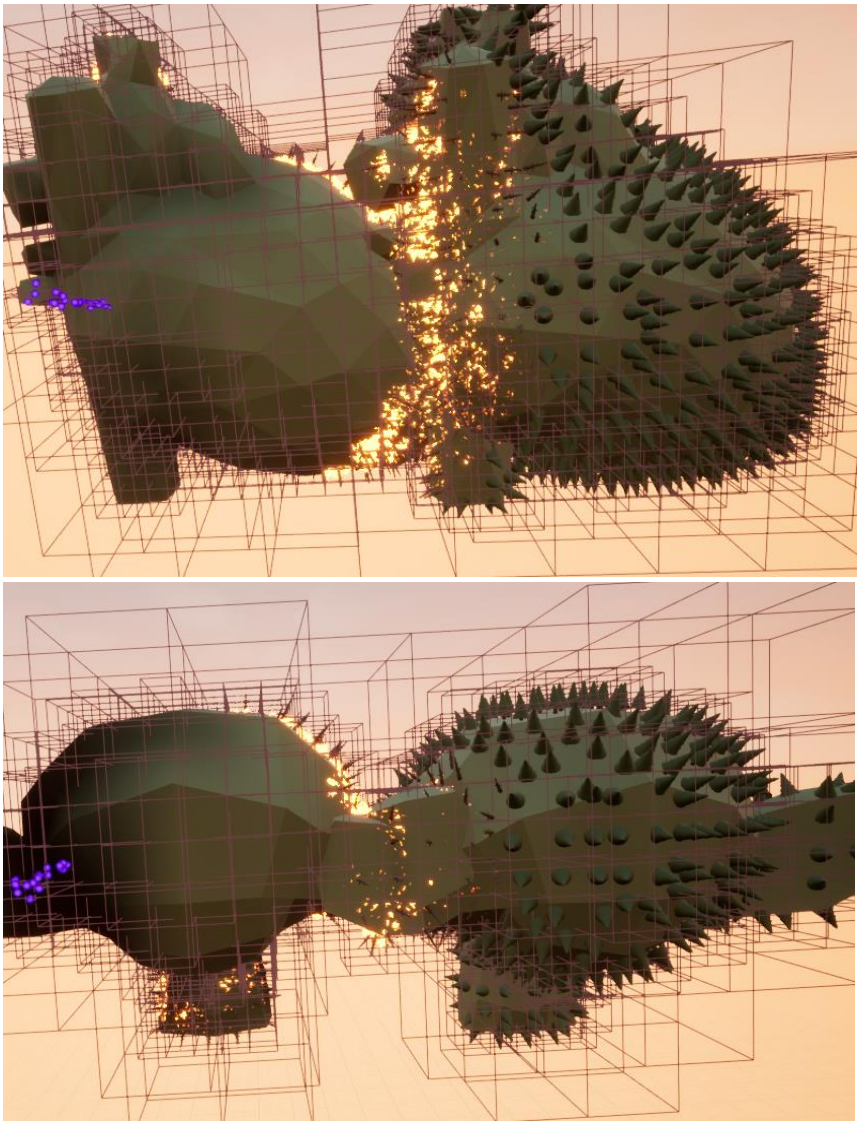


Fig. 13 Simulations running on terrain that exists out of multiple meshes (hearts and monkey heads).

One thing that could make the behavior of the fire propagation seem strange, is that when meshes overlap, the grid will still generate cells for the part of a terrain mesh that is inside another terrain mesh. This does not seem to cause trouble with the meshes that were tested, but could cause oddities with other, more complex meshes or combinations of meshes.

DISCUSSION

In the next sections, the data gathered for performance and the visuals from testing the algorithms will be looked into further.

PERFORMANCE

As the data in Table 1 shows, the VFX cost a lot of fps. Given that the default fps in the editor is 120, even with the smallest number of cells (674) in these test cases, the application loses 63 fps just by having to run the Niagara effects (Niagara is UE4's VFX editor). Still, considering that there are as many instances of VFX running as there are cells, these results are quite good.
The option to disable any activation of VFX is included in the application to prevent the fps from dropping too low when working with high numbers of cells.

When looking purely at the data from a single cycle, it is mainly the VFX that pushes the fps down. On average, turning on the VFX causes a drop of around 21 fps. The ticking of the cells also costs fps: starting the simulation makes fps drop. Compared to when all of the effects were on and cells were not ticking, even for the highest count of child cells, there still is a gain of 21 fps when doing a normal cycle with VFX and a gain of 34 fps without VFX.

If the simulation is run over multiple cycles using the options to regrow trees and pulse the ignition, the fps tends to be the same. When using VFX, there is a slight average increase of 2 fps when compared with the one cycle simulation. Without VFX, the same comparison gives a more obvious increase of around 10 fps. When observing the application run, its first cycle behaves much like in the previous test, reaching the same fps. However, as one leaves the application to run, the number of trees and empty cells evens itself out, causing the increase in fps. The fps is overall much more stable once running for more than one cycle, as there are no clear "completely overgrown" and "completely burned down" states.
When comparing VFX and no VFX over multiple cycles, the average drop caused by the effects is around 29 fps. Note that, in Table 3 and Chart 3, the first resulting fps without VFX would probably be higher than 120, as the engine caps its fps at that value.

When comparing the results of the build with what the editor brought up, the fps tends to converge the more detailed the grid is, as seen in Chart 4 and Chart 5. This could be due to the dynamic lighting of the trees, as the lighting quality is higher and thus the more trees, the more this will weigh down the fps. Another thing that stood out when comparing editor and build, is the fact that there haven't been any freezes or moments where the grid did not correctly find overlaps with the terrain, causing it to delete itself entirely. This probably has to do with the overhead when running in the editor, which might cause the overlap update to not run or run too late.

Overall, the build keeps performing better, even if the quality of the visuals is higher. With VFX, the increase was an average of 12 fps, while without VFX, the increase was about 26 fps.

GRID GENERATION AND INTERSECTION POINT ALGORITHMS

These algorithms are a bit harder to test, as they are mostly checked by simply looking at them. On top of that, the intersection point algorithm has a purely visual purpose, namely placing and orienting trees and fire effects.

After the tests with the monkey head and the heart, the algorithms seem to take care of generating the grid and finding intersection points without problem, as long as some limitations are kept in mind:

- Shapes should not be too thin, as this will cause two opposite sides to easily be encapsulated in one child cell.
- Double sided models are generally a bad idea. They will throw off the intersection algorithm.
- Very complex geometry or geometry that folds in on itself can cause some visual oddities when the grid is not detailed enough.
- When using multiple models, a grid will still be generated for the part of a mesh that is inside the other mesh.

The first three points do not influence the fire propagation or grid generation. They do influence how trees and fire are placed, and might cause odd locations and orientations to be present. When it comes to the grid generating for parts that are inside a mesh, in the test cases, this did not prove to be a problem. However, when using more complex geometry or more then two models, it could cause the fire propagation to behave wrong.

Overall, the grid generation and intersection point calculation work as intended, even with multiple meshes. Of course, some minor problems remain – namely some trees clipping through terrain and/or floating depending on the geometry the child cell covers. The more detailed the grid is, the less these oddities occur, but of course the heavier the simulation becomes. As long as the limitations are kept in mind, either by avoiding them or making the grid detailed enough, the simulator will provide good visuals.

CONCLUSION & FUTURE WORK

At the start of this paper, properties of a wildfire and methods used by other simulators were looked into, however, only some of them ended up being used in the application. Below, the properties and methods are again briefly touched upon, and the way they have been implemented or the reason they were left out is explained.

REAL FIRE VS SIMULATED FIRE

From all the properties of a real fire, only a few ended up being used in the actual simulator. This can be because they are expected to be “true”, but this can also be caused by certain traits not making sense within the environment the simulator creates.

THE FIRE TRIANGLE

Oxygen, heat and fuel are expected to be present. There is no option to alter the oxygen levels, which would make no sense. As for heat, one could say this is simulated by the ignition point, which could represent an electric spark, a lit match, or a lightning strike; anything that generates enough heat to

start a fire. Cells have fuel by default, and although the fuel amount can be changed, it is not currently possible to have a cell not burn at all.

WIND

Wind in the simulator is used much like the blog of Far Cry 2 [15] explains. The dot product of the normalized fire direction and wind direction is taken and applied to the fire damage. The wind direction vector is not normalized, as the length of the vector defines the speed or strength of the wind. In order to not deal negative damage, the final result is checked. If it is lower than 1, the damage is set to 1. This means that a fire *will* spread downwind, but at a much slower rate.
Real wind will constantly vary in strength and direction. In the simulation, the wind is static, always points in the same direction with the same strength.

FLAMMAGENITUS CLOUDS

Cloud formation is not used in the simulation. This also means there will be no influence of the fire on the weather system, and thus no possible rain, storm winds or new ignition points from lightning strikes.

SLOPE

Slope could have been implemented in a similar way as wind, however, due to the full 3D terrain the application uses, there is no reason for this. There is no up and down, no polygons that can be seen as "sloping" in comparison to the others. Perhaps one could use the difference in direction between he calculated intersection normal and the normal of any of the enveloped planes, but even so, it would give odd results.

FUEL

In the application, only the amount of fuel and the fire resistance play a role. The combination of these two can simulate different types and sizes of fuel and represent moisture. Of course, as the application looks at large-scale fire and not at the detailed spread of wildfire, details of fire spread will get lost. There will be no way of seeing exactly how the fire spread through a cell, what plants burned first and what burned the longest. This level of detail, within the idea of the application, is simply not needed.

CROWN, SURFACE AND GROUND FIRE

The application makes no difference between the different kinds of fire, as again, it is not needed for what the application is meant to do.

CELLS OR WAVELETS

Simulators that use Huygens principle are simulators which need a lot of time to calculate and render their final result. The mathematics behind it are simply too complex to be done at runtime. These simulators often also have the capability to keep track of fuel types, changing weather conditions etc.,

making them powerful but slow. [9], [11]
Calculations based on cells, especially when the workings of the wildfire have been simplified to fit the needs of a non-realistic fire, are much faster. Cell2Fire is already faster than Prometheus and FARSITE, while the Far Cry 2 simulator is definitely the simplest and fastest of them all [9], [11], [13].
This is also why the wildfire simulator application uses cells to propagate the fire. It is faster, capable of doing wave propagation at low cost, and it leaves room to run the actual visualization next to it.

SOFTWARE

Unreal Engine 4 is definitely a handy tool to make these kind of programs. As the trees are generated at runtime, their lighting is also calculated at runtime. As long as not too many light sources are used, UE4 is perfectly capable of running this. If one would want to optimize it more, the option to not cast shadows could be used on the trees. This will cause some visual difference of course, but will save fps. Using the optimization tools such as the option to view shader complexity helps to optimize the VFX and figure out where any bottlenecks are.
On top of that, the engine provides a lot of functionality that is useful in a simulator like this. They are not necessarily engine-only, but save time as one does not have to implement it themselves. Examples of this are the functionality behind the tick, the option to do overlaps with certain actors in certain shapes, and tools like Niagara and the widget editor. Blueprint can also be handy to quickly test out things.

However, Unreal Engine is of course *too much* for an application like this, and optimizations might hide in the source code itself. Dedicated engines, build specifically for a wildfire simulator, might be able to get higher fps and still do more calculations, for example when adding a weather system.

WILDFIRE VS CARDIAC EXCITATION

When it comes to visualizing cardiac excitation, it certainly is possible, perhaps if simplification is allowed. However, in order to generate all of the patterns, many more options have to be included. Currently, it should be possible to create a normal rhythm by tweaking the values for fuel, resistance, and tick rate. Abnormal rhythms will, however, need more features to be implemented. Things like multiple ignition points, obstacles, ignition points of different shapes etc. could all help achieving these rhythms.

FUTURE WORK

The algorithms to generate the grid and find the intersection points do their job well, but more advanced ones could get rid of the few remaining problems.

The UI works, but is rough and often a certain order of input is needed to get something working as one wants to. This is the place where the application could use the most work: in order to make an actual simulation program out of it, what could be added is (apart from better visuals):

- A place to upload custom meshes

- Choose from meshes included in the app and/or custom ones to be the terrain.
- If going further into that, even a simple gizmo could be included, to let the user place the meshes around like they want to.
- Grid generation works, but it often resets values. On top of that, it could be good for performance to be able to locally create grids. For example, generating the parent grid, and when selecting a parent cell, having the option to generate a child grid only in that one cell. This way, a grid can be as detailed as needed for the underlying geometry.
- Turning the grid visualization on and off.
- Turning the tree visualization on and off.
- Selecting cells feels tedious as one can only select one (parent) cell at a time. Having a system that lets you choose between selecting parent and child cells and being able to select multiple at once would help a lot.
- Ignition currently always happens in one point. However, being able to start a fire in multiple places, possibly with user-defined delays, could prove to be really interesting. With that, one could also implement ignition lines, where a fire starts as front instead of a single point.
- Better gizmo/compass for setting wind direction and speed. Right now, it takes a vector, which works, but it's a game of guessing as to what you will visually end up with until all values have been inputted.
- The ability to instantly put out a fire during simulation.
- The ability to instantly set a cell on fire during simulation.
- Placing of obstacles.

This list is non-exhaustive, as tools like this always can have something added to it to make it just that little bit better.
Of course, many of these require functionality on the simulation side as well, and are on their own not too hard to implement. The problem - and main time-consumer when it comes to implementing all the options - lies in the variety of options on a click and/or drag from the user, and making sure nothing collides with anything else.

BIBLIOGRAPHY

[1] "Wildfires: Why they start and how they can be stopped - BBC News." https://www.bbc.com/news/newsbeat-41608281 (accessed Dec. 16, 2021).

[2] "Weather Elements." https://www.auburn.edu/academic/forestry_wildlife/fire/weather_elements.htm (accessed Dec. 16, 2021).

[3] "Flammagenitus cloud - Wikipedia." https://en.wikipedia.org/wiki/Flammagenitus_cloud (accessed Dec. 17, 2021).

[4] T. Hädrich, D. T. Banuti, W. Pałubicki, S. Pirk, and D. L. Michels, "Fire in paradise," *ACM Transactions on Graphics*, vol. 40, no. 4, Jul. 2021, doi: 10.1145/3450626.3459954.

[5] "Flammagenitus Clouds: Pyrocumulus Formation | WhatsThisCloud." https://whatsthiscloud.com/other-clouds/flammagenitus/ (accessed Dec. 17, 2021).

[6] "Cumulonimbus flammagenitus - Wikipedia." https://en.wikipedia.org/wiki/Cumulonimbus_flammagenitus (accessed Dec. 16, 2021).

[7] "Topography." http://www.auburn.edu/academic/forestry_wildlife/fire/topos_effect.htm (accessed Dec. 16, 2021).

[8] "Fuel's Effect on Fire." http://www.auburn.edu/academic/forestry_wildlife/fire/fuels_effect.htm (accessed Dec. 16, 2021).

[9] C. Tymstra, R. W. Bryce, B. M. Wotton, S. W. Taylor, and O. B. Armitage, "Development and Structure of Prometheus: the Canadian Wildland Fire Growth Simulation Model."

[10] "The Different Types of Wildland Fires." https://www.supplycache.com/blogs/news/the-different-types-of-wildland-fires (accessed Dec. 16, 2021).

[11] M. A. Finney, M. Alexander, P. Andrews, J. Beck, B. Keane, and J. Scott, "FARSITE: Fire Area Simulator-Model Development and Evaluation," 1998, Accessed: Dec. 17, 2021. [Online]. Available: http://www.farsite.org

[12] "FARSITE | Rocky Mountain Research Station." https://www.fs.usda.gov/rmrs/tools/farsite (accessed Dec. 17, 2021).

[13] C. Pais, J. Carrasco, D. L. Martell, A. Weintraub, and D. L. Woodruff, "Cell2Fire: A Cell Based Forest Fire Growth Model," May 2019, [Online]. Available: http://arxiv.org/abs/1905.09317

[14] "Wave Interference | CK-12 Foundation." https://flexbooks.ck12.org/cbook/ck-12-middle-school-physical-science-flexbook-2.0/section/16.10/primary/lesson/wave-interference-ms-ps/# (accessed Dec. 17, 2021).

[15] "Far Cry: How the Fire Burns and Spreads | Jean-Francois Levesque." http://jflevesque.com/2012/12/06/far-cry-how-the-fire-burns-and-spreads/ (accessed Oct. 29, 2021).

[16] "Niagara : How to create a stylized FIRE Effect - UE4 tutorials [Distortion - Opacity Mask] - YouTube." https://www.youtube.com/watch?v=hivNxPXG-Mg (accessed Jan. 10, 2022).

[17] "3D Printable Low Poly Heart Model by Girolamo Caiazzo." https://www.myminifactory.com/object/3d-print-low-poly-heart-model-17173 (accessed Jan. 10, 2022).

[18] "SAFERS PROJECT - Homepage." https://safers-project.eu/ (accessed Dec. 17, 2021).

[19] "Wildland Fire Facts: There Must Be All Three (U.S. National Park Service)." https://www.nps.gov/articles/wildlandfire-facts-fuel-heat-oxygen.htm (accessed Dec. 17, 2021).

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF CHARTS