White Paper

Intel Software and Services Group

Hugh Smith

Procedural Trees and Procedural Fire in a Virtual World

October 2008

Intel Corporation

Contents

1 Introduction

The Tree and Fire Systems are part of the Intel Smoke Framework, a multi-threaded game engine. Procedurally generated trees and procedurally generated fire were inspired by the need to generate a virtual burning forest.

To animate a burning forest, you must illustrate many natural variables. To be convincing, procedurally generated content must be predictable and consistent, yet must also have believable variation. This paper explores the implementation of the Tree and Fire Systems for procedurally generated trees and the procedurally generated fire. It also describes the creation and implementation of the grammar and parser used to create the trees, as well as examining the Fire System which uses the Smart Particle Engine to set the trees on fire.

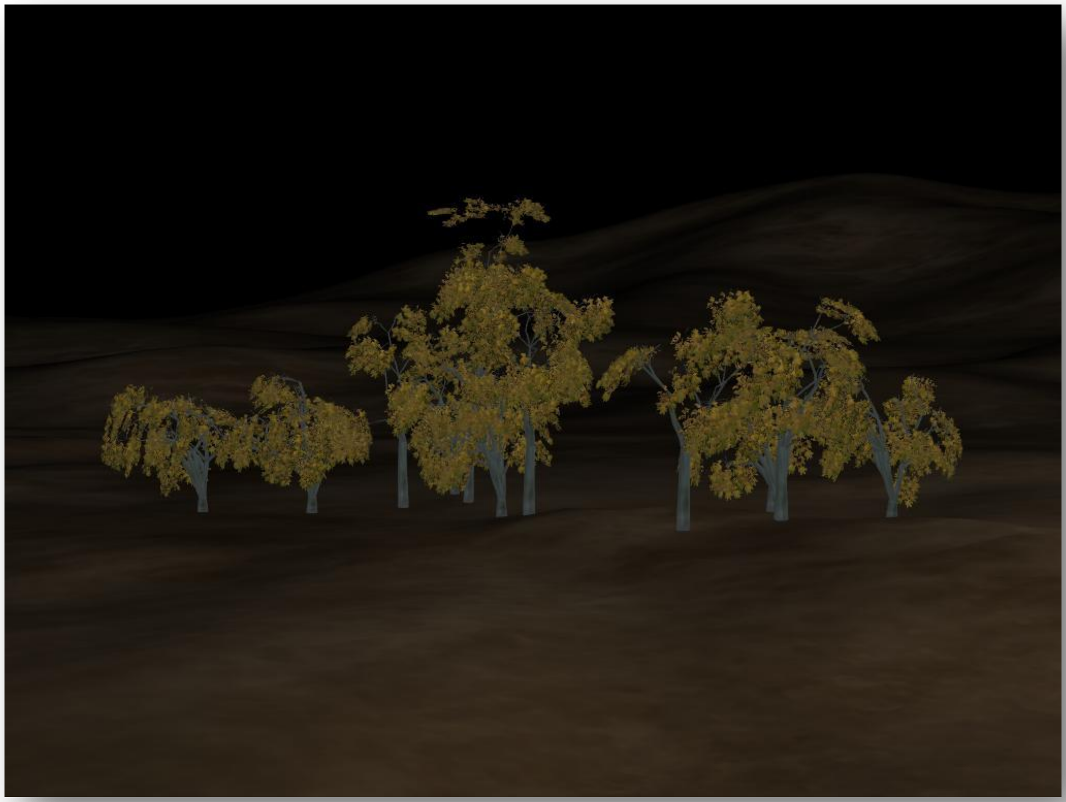


Figure 1 The Forest

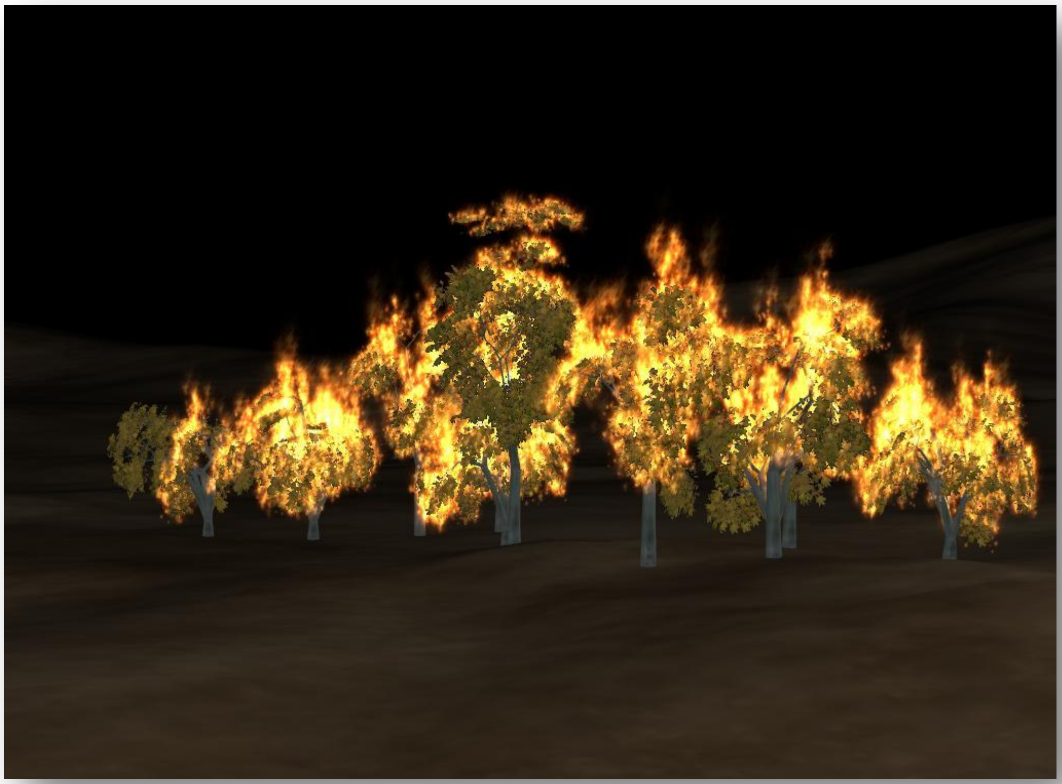


Figure 2 The Forest on Fire

2 Trees

Trees are naturally complex. Nature has given us what seems to be a never ending array of tree species and foliage. As a developer, when simulating trees with a procedural methodology, you must consider what differentiates a tree from a bush and one tree from another. You must also familiarize yourself with the tree structure including how the branches join the trunk and how the canopy affects the overall shape of the tree. How do you create algorithms and structures that can capture the essence of trees? All trees are similar but no two trees are alike, so your algorithm must provide believable variation. How do you create an algorithm that will procedurally create similar but not identical instances of trees? This is the world of simulating natural vegetation and this section explores how to create procedurally generated trees. There are many resources available about graphically simulating natural vegetation, but one stands out: "The Algorithmic Beauty of Plants" by Przemyslaw Prusinkiewicz and Aristid Lindenmayer [1]. This paper is recommended for anyone interested in exploring this topic in more depth.

2.1 Tree Properties and Fire

To create a burning tree, you must consider the tree in detail. Creating individual trees in a modeling program with an artist is one way to get good looking trees, but this method limits the number of trees you can create and the result is just a basic model of vertices and triangles. A modeled tree looks like a tree but it does not contain the necessary context or structural data you need to associate it with a context sensitive fire. If you are going to set your tree ablaze, unless you want to paste a generic fire particle engine on each tree, you need more information about the tree.

For each tree or group of trees you must consider how the behavior of fire is affected by the structure of the trees. For example:

- fire spreads - it spreads upwards and outwards, and spreads from branch to branch based on fuel and locality
- fire forms to the fuel - it takes the shape of the fuel source, and because each tree is different the fire should form to it differently

With these properties in mind, the need to generate trees in a procedural manner is apparent.

2.2 Tree Parser and Grammar

To create procedural trees a general tree parser was implemented that when given a grammar describing a tree species, produces a specimen tree of that grammar type. Using the parser is as simple as sending the parser a grammar and a random seed; it creates a unique deterministic tree based on your input. Plant a seed and grow a tree.

The tree parser is a separate external library used by the Tree and Fire Systems projects. This is to ensure that you can easily use the tree parser library in separate projects as needed or desired.

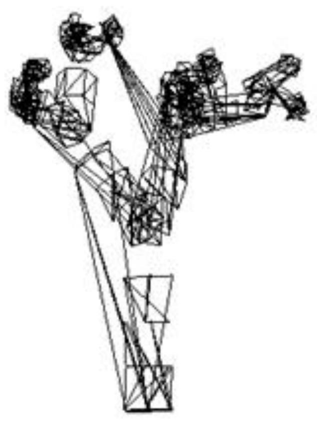


Figure 3 No Experiment Goes Without Hitches: The First Rendered Tree

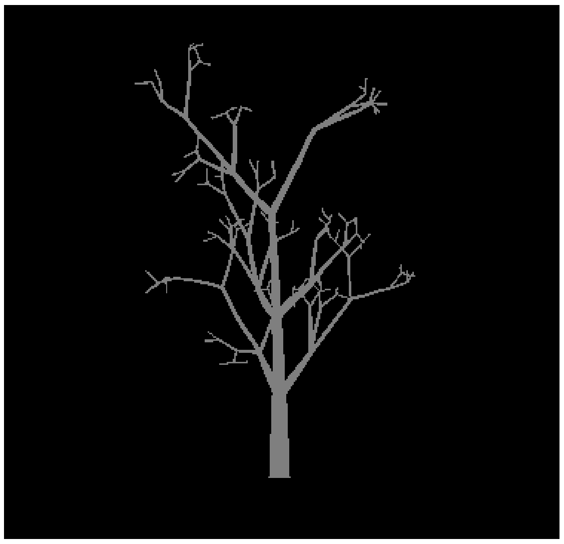


Figure 4 The First Reasonable Looking Tree

3 Grammar

In the context of this paper, a grammar is the set of rules defined to tell the parser how to interpret the token stream. The set of rules, or individual grammar, produces different instances of a specific species of tree depending on your input. For the Tree and Fire Systems the grammar is based on the concept of a Turtle or L-System grammar model.

Looking at a tree you can see that it has characteristics that are specific to being a tree and characteristics that are specific to being a species of tree. Although the parser is generic in that it will parse and create a tree given any well formed grammar describing the tree, it is only a tree parser; that is, it will not take a grammar and produce a house or a car. As a result, the parser has very specific grammar requirements to ensure consistency of the trees. When creating a grammar you must consider these characteristics of a tree:

- Trees have trunks
- Trunks have splits - branches split off from the trunk
    - Splits have levels - there may be multiple branches splitting off at the same place
    - Splits have drop angles - each branch has a drop angle at the split
    - Trunks have a heading - the main trunk might continue straight after a split
- Trees have branches
    - Branches can vary in length
    - Branches can be straight, curved, or even wobble
- Trees have a canopy
    - Canopies have a general arch
    - Canopies tend to be more concentrated at the ends of the branches

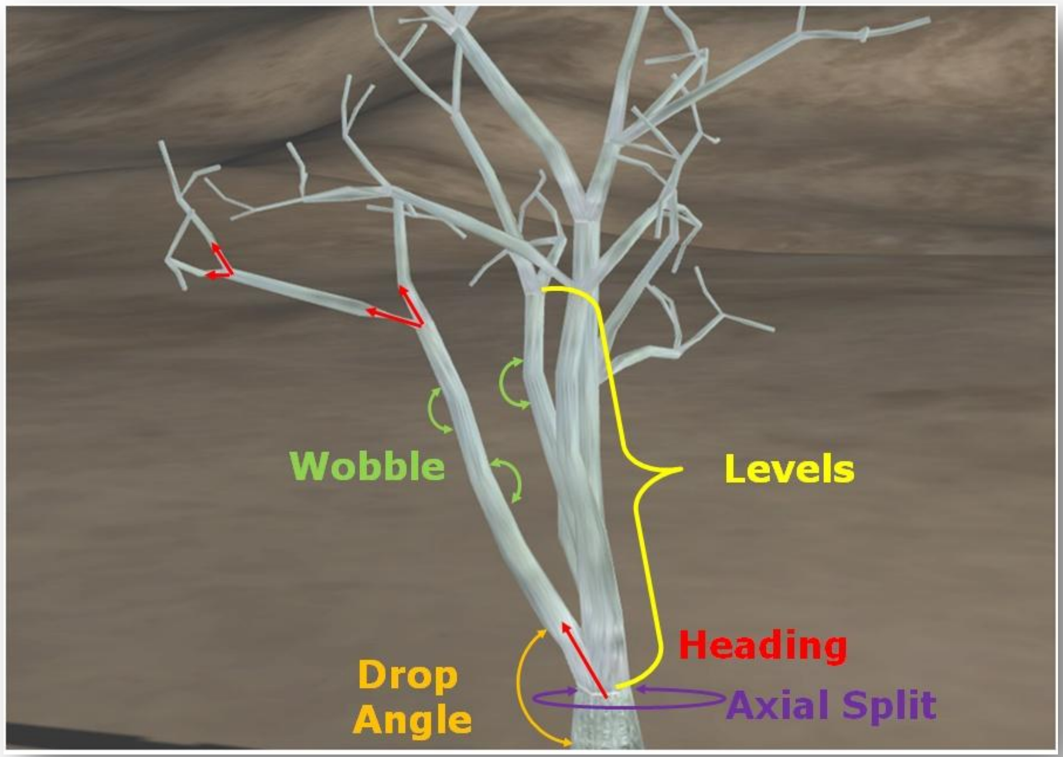


Figure 5 Characteristics to Consider in a Grammar

3.1 Level Grammar

Trees are broken into different levels, and each level has its own grammar. That grammar defines the characteristics specific to each level, such as how much each branch wobbles and how far the branch drops from the parent branch when it splits off. You also use grammar to specify the number of levels you want to define. As you experiment and change the parameters for each level and each variable, you change the characteristics of the trees.

3.2 Canopy Grammar

When designing the canopy you must consider issues such as the angle that the canopy hangs or falls, the size of the canopy pieces, and the location of the fulcrum of the overall arch of the canopy. You can define these factors either in the grammar or the parser. Given development time constraints in this version of the Tree library you will find that certain canopy characteristics, such as the fulcrum point, canopy patch height, and canopy patch width are hardcoded into the parser. In practice, these characteristics should be moved to the grammar for more flexibility.

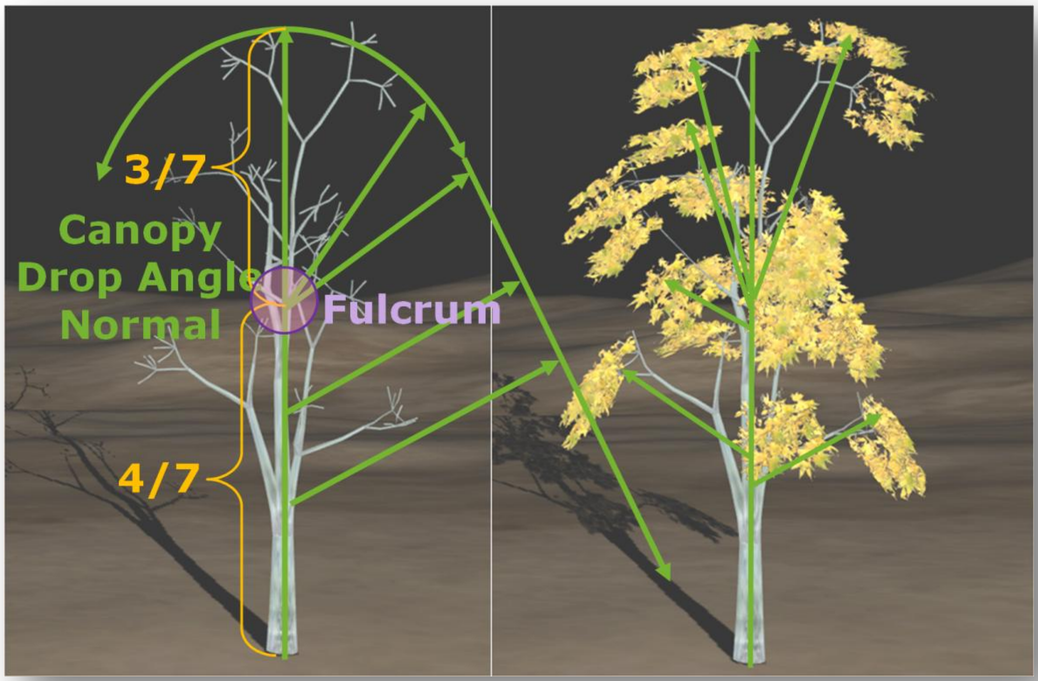


Figure 6 Characteristics of the Canopy

3.3 Split Grammar

How trees split is important. Some trees have splits where you can see each level distinctly. Some splits seem to repeat at each level. Other splits seem to be a continuation of the previous branch, such as the main trunk or major branch members. These split characteristics can be mimicked in the grammar. In the case of splits, split buckets are defined in the grammar as the probability of getting one bucket versus getting another. Then when the parser generates a random number token for that split level it determines what pattern that tree will follow. There are no rules as to how many or how few buckets you can create; the only requirement is that the buckets add up to 100 percent, as described in the following paragraphs.

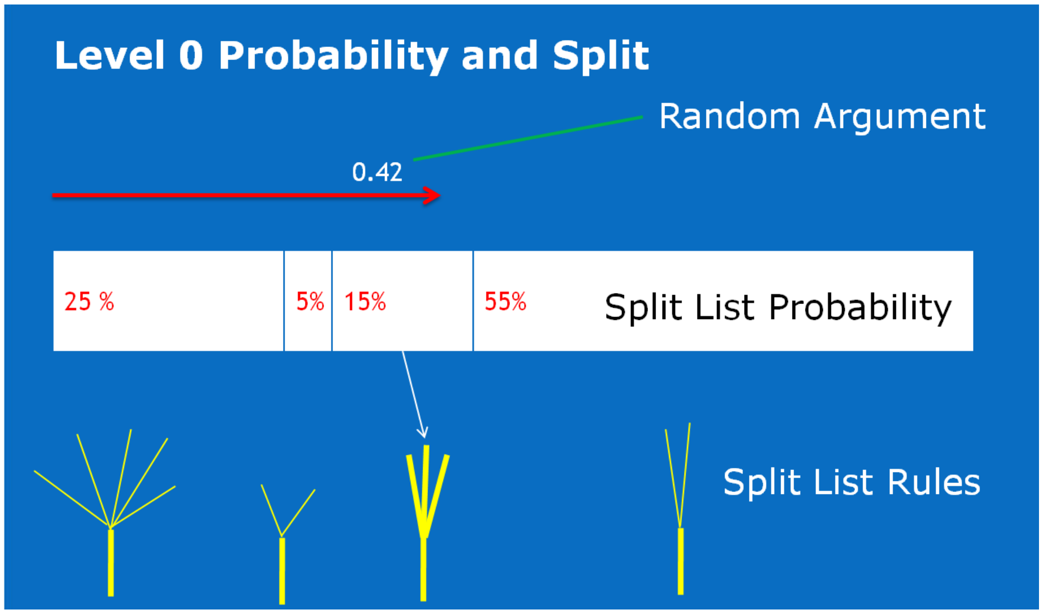


Figure 7 Split List for a Level of a Grammar with Four Split Options

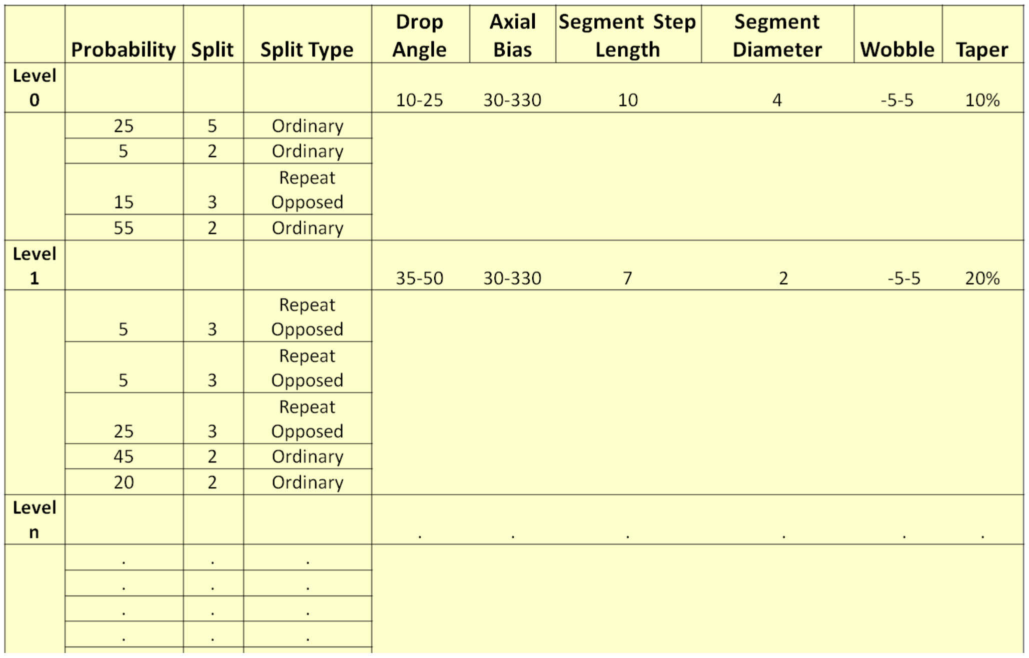
For the example in the diagram above the randomArg is between 0 and 1. The probabilities of the splitList array elements will add to 1.0, where 1.0 represents 100 percent. For example, if there are four split options with probabilities of 0.25, 0.5, 0.15, and 0.55, respectively, then there is a total of 100 percent. The randomArg generated for each split will lie somewhere in one of the four ranges. So, if you have a randomArg value of 0.42 that means that the third split, or the one with the probability of 0.15 will be chosen. That is, 0.25 + 0.5 + 0.15 = 0.45, and 0.30 < 0.42 < = 0.45.

Comparing the illustration in

Figure 7 to the Level 0 row of

Figure 8, you can see Level 0 contains options for the splits in

Figure 7. The first split possibility indicates that the tree could have five splits. The third split possibility could create three splits, but the grammar level repeats itself. Conveniently, you can define the grammar any way you want. Split categories, split count per category, and type of split are defined in the grammar. You define the splits for each level. For an example of a full grammar, refer to Appendix A Grammar and Parser Details on page 35.



	Probability	Split	Split Type	Drop Angle	Axial Bias	Segment Step Length	Segment Diameter	Wobble	Taper
Level 0				10-25	30-330	10	4	-5-5	10%
	25	5	Ordinary						
	5	2	Ordinary						
	15	3	Repeat Opposed						
	55	2	Ordinary						
Level 1				35-50	30-330	7	2	-5-5	20%
	5	3	Repeat Opposed						
	5	3	Repeat Opposed						
	25	3	Repeat Opposed						
	45	2	Ordinary						
	20	2	Ordinary						
Level n				.	.	.	.	.	.
	.	.	.						
	.	.	.						
	.	.	.						
	.	.	.						

Figure 8 Basic Structure of a Grammar

The table in

Figure 8 illustrates some of the above grammar concepts.

For this library, to ensure the grammar can be edited by the artist or programmer the grammars for each tree species are XML based. The parser uses tinyXml to read the XML grammar file. It then tokenizes the grammar into an internal grammar structure which is used by the tree parser.

4 Parser

The parser takes the grammar and, along with a seed for the random generator, creates an individual tree. The parser is the code that makes up the tree library. It is where the core of the implementation for the procedural trees resides. The tree structure it produces is a linked list of branches (the treeNode). The branch structure (branchBase) contains a list of segment points (vertex points), heading, position, Vertex Count, AABB (Axis Aligned Bounding Box), life, and burning. Note that the life and the burning variables are not currently implemented but are included for completeness.

```
struct treeNode {
   branchBase *pbranch;
   treeNode *pPrevNode;
   treeNode *pNextNodes;
   int nextNodeSize;
   Tree *tree;
};

struct branchBase
{
   segment *segments; //create a list of segments for this branch
   canopy  *canopies;
   bool isCanopy;
   int segmentCount;
    int tipPointCount;
   V3 heading; // This is overall branch direction
   V3 position; // This is the root segment center point.
   V3 tipPoint; // This is the last segment center point.
   branchType type;
   int startVertex;
   int startIndex;
   int vertexCount;
   int indexCount;
   aabb AABB;
   long life; // here for completeness not implemented
   bool burning; // here for completeness not implemented
};
```

The following graphics explain what is meant by segments and branch structures.

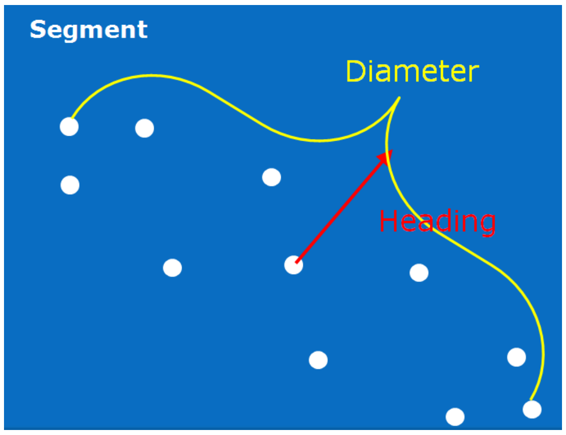


Figure 9 Segment of a Branch is the Ring of Points with a Heading

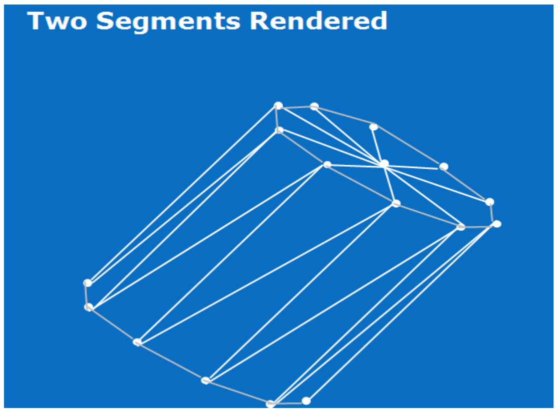


Figure 10 Two Segments

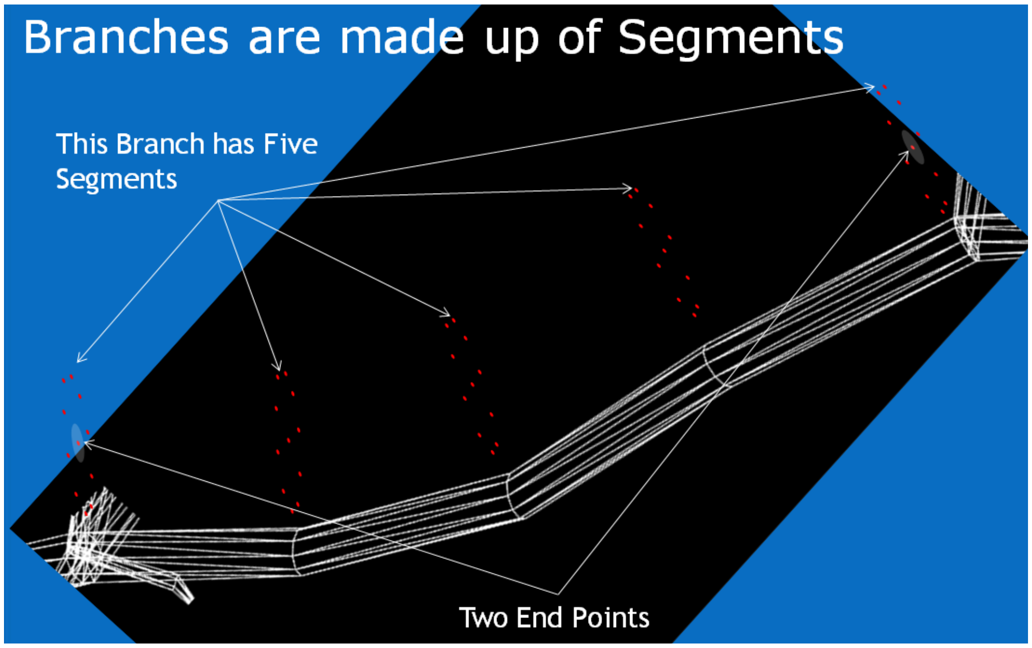

Branches are made up of Segments
This Branch has Five Segments
Two End Points

Figure 11 Branch, Segments, and End Points

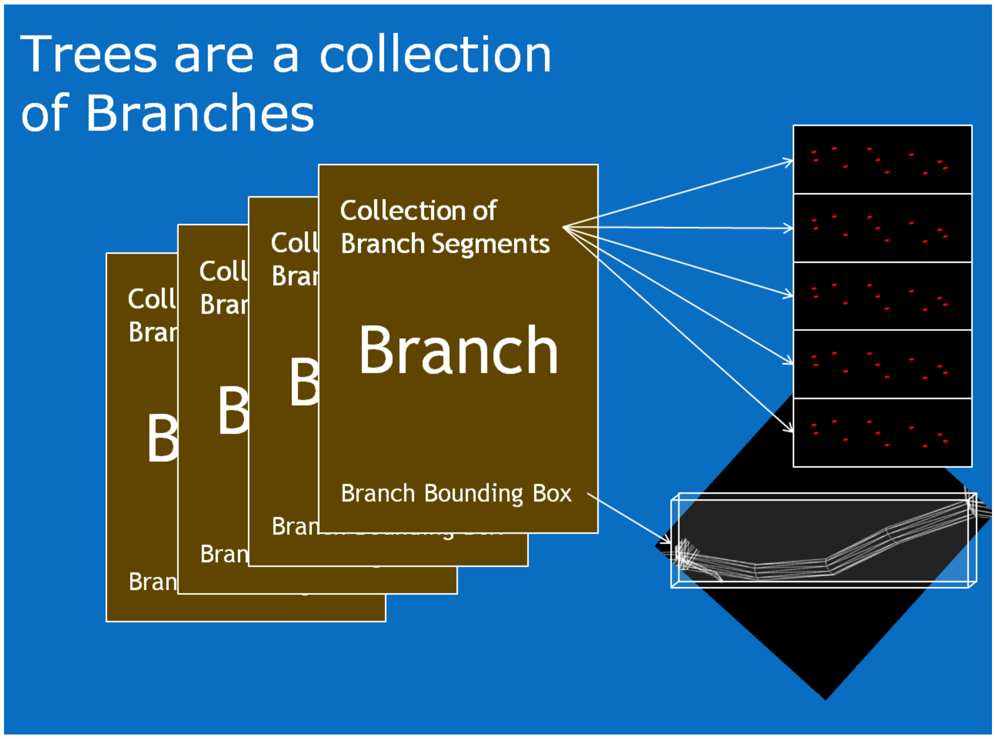

Trees are a collection of Branches
Collection of Branch Segments
Branch
Branch Bounding Box

Figure 12 Collection of Branch Structures for a Tree

When an instance of a tree is created, a linked list of branches and canopies is procedurally generated. The nodeTree is this collection of branches and canopies in the form of a linked list. This linked list, or nodeTree, is made up of treeNode structures.

Note: nodeTree is the single instance linked list per tree, and the treeNode is the structure containing branch and canopy data that has many instances per tree. The treeNode is stored individually at each node of the linked list.

Once the nodeTree is completed with all the branch and canopy data it can be traversed to collect the information needed when producing the renderable model.

Example: Simple Sequence of Creating and Filling the Model

1. Create a treeNode called Trunk.
2. Call the recursive function growBranch with the parent, node structure, grammar and a beginning heading.
   Note: (0,1,0) is a good initial heading as trees generally grow upwards.

   ```
   Trunk->growBranch(Trunk, treeNode, &grammar, startHeading);
   growBranch(…) {
     calculates the segments for the branch
     fills the branch structure
     adds the branch to the treeNode linked list
     calculates the splits and split types based on the grammar
     for each split calls growBranch(…)
   }
   ```
3. Once the branches are complete, you then do a similar recursive pass for the canopy. The canopy is made up of canopy segments much like branch segments.

   ```
   growCanopy(nodeTree)
   ```

   The function is a recursive call much like growBranch.

   ```
   growCanopy(…){
   determine if branch is of canopy type
   growCanopyPatch
   growCanopy(nextNode)
   }struct RenderStructure {
      VertexType type;
      VertexPos *vpos;
      VertexPNT *vpnt;
      WORD *ptrIBData;
      DWORD CurrentIndex;
      DWORD CurrentVIndex;
        WORD BranchCount;
      bool ReverseWindingOrder;
      aabb AABB;
   };
   ```

   Now the basic nodeTree linked list of branches and canopies is filled with branch and canopy data.
4. Next you create a model that can be rendered by filling out the vertex and index buffers. In the application that uses the tree library you will create a vertex buffer and index buffer to hold the data for your renderable tree(s). Use the RenderStructure as described in the following steps to accomplish this.

   The process can be explained this way: when producing a renderable model of the tree, you start with the populated nodeTree linked list and then pass the unpopulated vertex and index buffer into the parser. It will fill the buffers with the vertex information, index information, and vertex declaration information. Presently the vertex declaration is limited to just a few items: position, texcoords, and normals. However you could easily extend the parser code to include bi-tangent coordinates or anything else that would be of interest. You now have what you need to render the trees. The RenderStructure contains the pointers to the Vertex and Index buffers that you use to store the data from the tree you just created.

5. Finally, you are ready to populate and use the RenderStructure. From the application you initialize the RenderStructure and then call the methods to fill the buffers. Keep in mind that the treeNode has already been created.

   To understand the example below the concept of the observer must be defined. While traversing linked lists in a recursive manner it is very difficult to keep track of all the states at any given time within any specific iteration of the recursion. Conceptually, the observer is a singleton container for global states: a type of controlled global variable. The parser uses the concept of an observer to help manage the state. It gives any recursive function the necessary visibility into the current position within the linked list while also allowing it to have the global scope needed for certain variables. The instance of the observer can be called at any time and the state is always consistent. The observer is used internally to the parser and is also exposed to the application using the tree library. The observer owns and controls the render structure. The application fills out the details of the RenderStructure needed and the parser then, with the help of the observer, fills the vertex and index structures that are referenced within.

   This is an example of the observer being used in the application:

```
Observer->DXRS->CurrentIndex        = 0;
 Observer->DXRS->CurrentVIndex      = 0;
 Observer->DXRS->BranchCount        = 0;
 Observer->DXRS->ReverseWindingOrder = true;
 Observer->DXRS->ptrIBData = reinterpret_cast<WORD*>(pIndices);
 Observer->DXRS->vpnt = reinterpret_cast<VertexPNT*>(pVertices);
 if(m_PrimitiveType == Primitive_Branches)
 {
    m_tree->fillBranches( m_tree->nodeTree );
 }else if(m_PrimitiveType == Primitive_Canopy)
 {
    m_tree->fillCanopies( m_tree->nodeTree );
 }
```

   The trees are now ready to render.

   In the figure below there are four tree grammars (TGmr1-TGmr 4) mapped with random seeds.

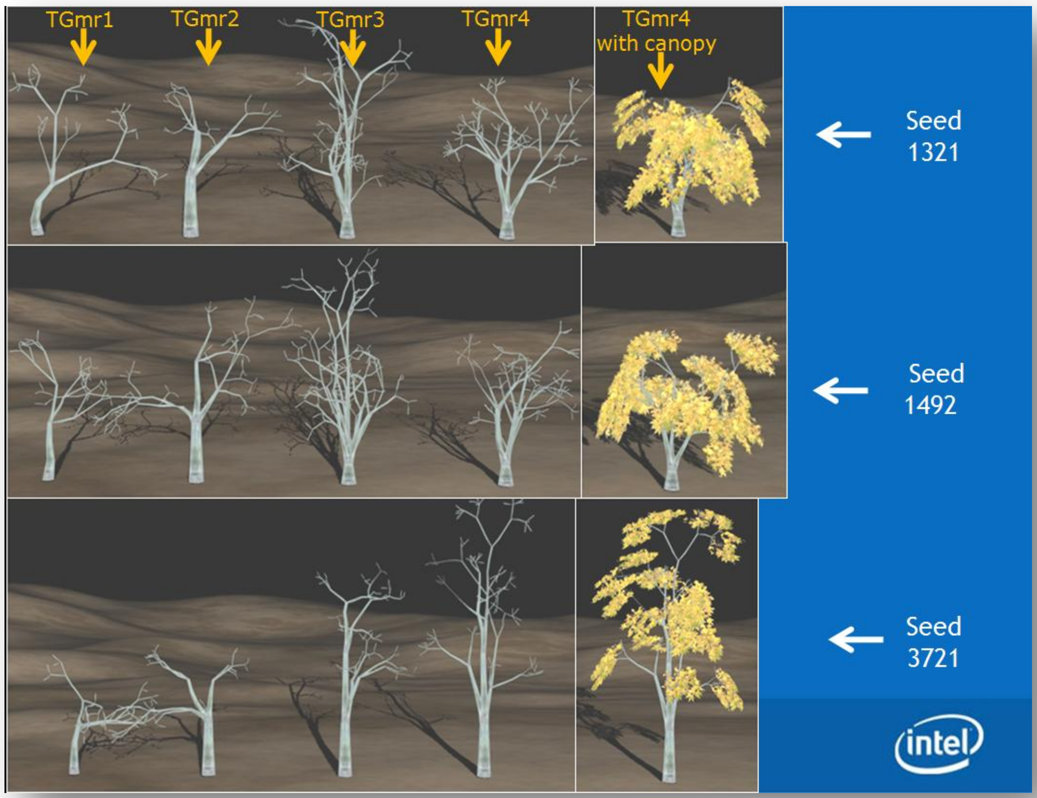


Figure 13 Grammars and Seeds

5 Procedural Fire

Procedural fire has a lot of implementation possibilities beyond simply burning down trees. When you want fire in your scene, the concept that one object on fire can set something else on fire opens the door to a lot of creative potential. For example, imagine a scene in which burning meteors set trees on fire. You might choose to make the fire spread by scripting it or by building a custom fire for each object and somehow synchronizing the events. But neither of these methods results in realistic-looking spreading fire. The goal for this fire system would be to create fire that is believable to watch as it spreads; it should be predictable but not deterministic.

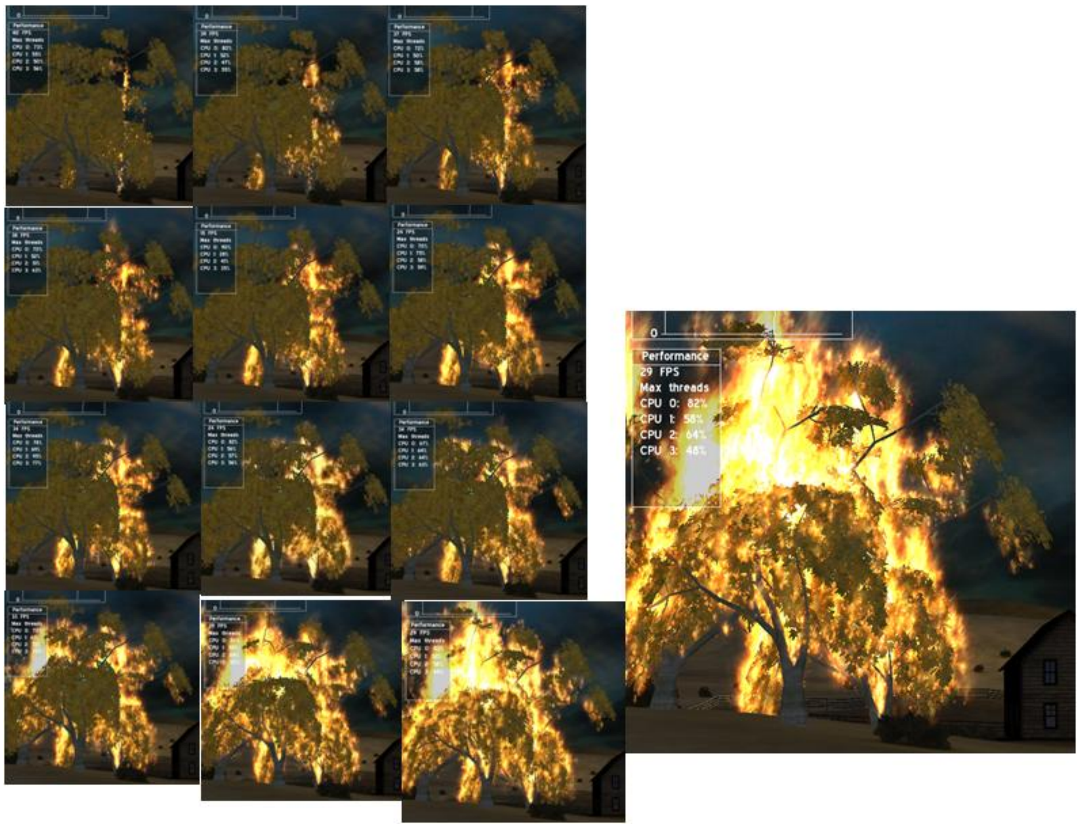
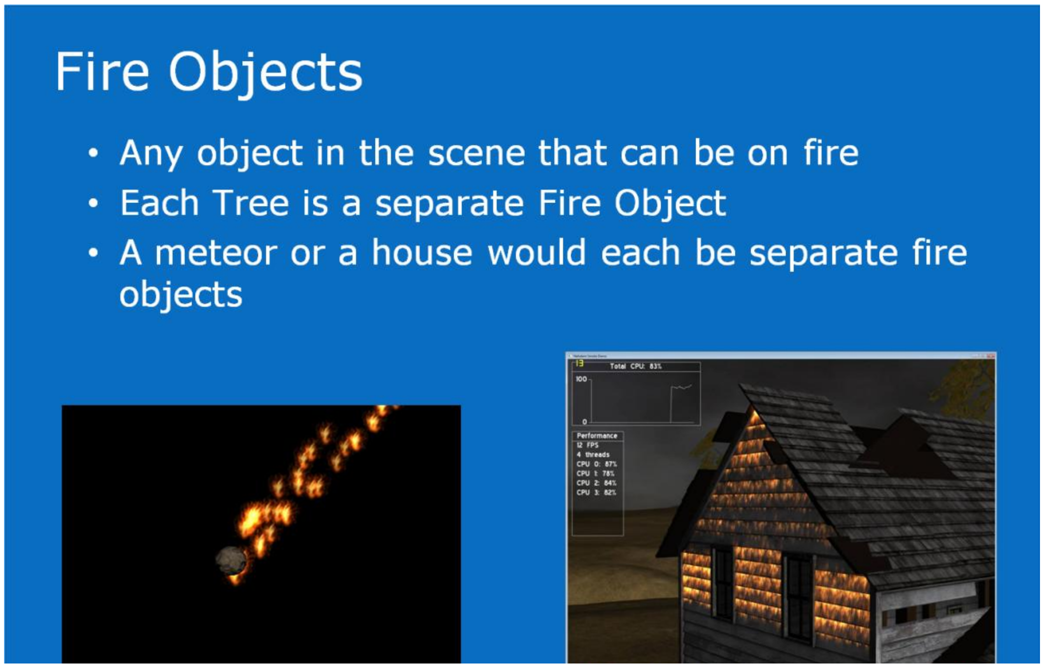


Figure 14 Fire Spreads on the Tree

Instead, the fire should be procedural in nature. This is made possible by the creation of what is referred to in this paper as a “Smart Particle Engine,” discussed in the following section.



Fire Objects

- Any object in the scene that can be on fire
- Each Tree is a separate Fire Object
- A meteor or a house would each be separate fire objects

```
13        Total  CPU:  83%
100
0
Performance
12  FPS
4  threads
CPU  0:  87%
CPU  1:  78%
CPU  2:  84%
CPU  3:  82%
```

Figure 15 Fire Objects

6 Smart Particle Engine

It is assumed that readers of this document have a basic understanding of particle engines. There are many good references on the topic including the example of a particle engine given by Frank Luna in his book “Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach” [4]. Although the particle engine that was derived for this project is hardly recognizable from the example given in the book, Frank Luna’s work should be acknowledged. It is an excellent starting point and provides some basic inspiration.

A smart particle engine has two elements: the visible piece and the non visible piece. The visible piece is what most people think of with a typical particle engine. In this case the visible piece is the flames; it is what tells the viewer that something is on fire. The second piece is the physical part that is not visible. This non-visible part has some physical properties. These properties could be one or more things such as heat, cold, gravity, and so on. For the fire system, the smart particles are used as heat. Every fire particle engine has an independent heat particle system embedded in it. These heat particles are not visible but they do act on the surrounding environment. The fire system is a collection of Fire Objects each of which has one or more smart particle engines. These smart particles are used to interact with the other objects in the scene. In this case, those objects are heat particles. For this paper, the topic has been narrowed only to the part of the fire system that interacts with the trees.

Smart Particle Engines are made up of two parts:

- Fire Particles

Figure 16 The Visible Part, What We Think of as Particles

- Heat Emitter Particles

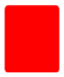


Figure 17 The Invisible Part, Physical in Nature

6.1 Spreading Fire: Particles and Emitters

The Tree is a Fire Object with many fires. In general each fire has its own smart particle engine. In the case of the Tree type Fire Object, each branch has a fire and its own particle engine. The Fire System contains several types of fires, spheres, patches, and in the case of a tree branch a “fireline” type particle engine. This is a particle engine that expresses particles along a given vector. In the case of the tree, each branch object has two endpoints (

Figure 11). These two points create the vector along which the fire for the branch is defined. If this engine is turned on, then the branch appears to be on fire.

One property of fire is that it spreads. This property is where the smart particles are especially useful. A particle engine is either active (visible) or inactive (not visible). When a Fire Object fire is active, for example a fire on a branch is visible, it also spawns its embedded heat particle engine. The heat particle engine acts as a heat emitter. By projecting these heat particles such that they mimic heat you can simulate a spreading fire. The heat particle travels along a trajectory, and you create a ray based on its current and previous positions. When a heat particle ray intersects the bounding box of another fire in a Fire Object, the particle engine for that intersected fire becomes active (visible) and in turn spawns its own heat emitter.

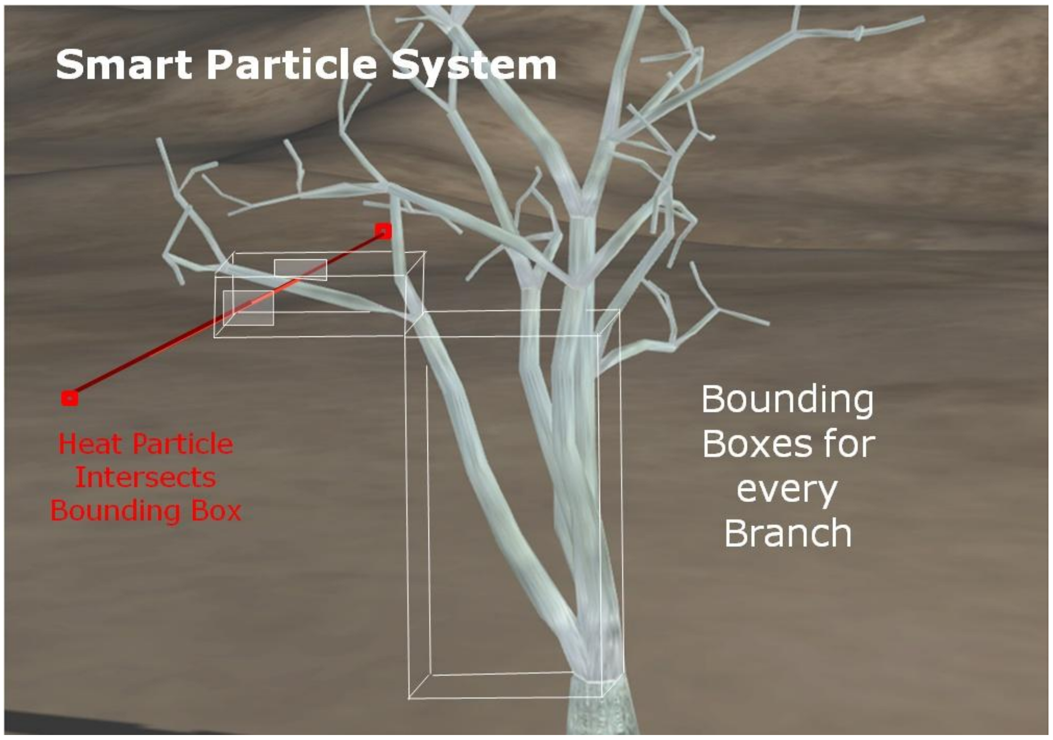

Smart Particle System
Heat Particle Intersects Bounding Box
Bounding Boxes for every Branch

Figure 18 The Vector of the Heat Particle Intersects the Bounding Box of a Branch

This behavior has some interesting effects: as more branches catch fire, more heat emitters become active, and the fire begins to spread faster.

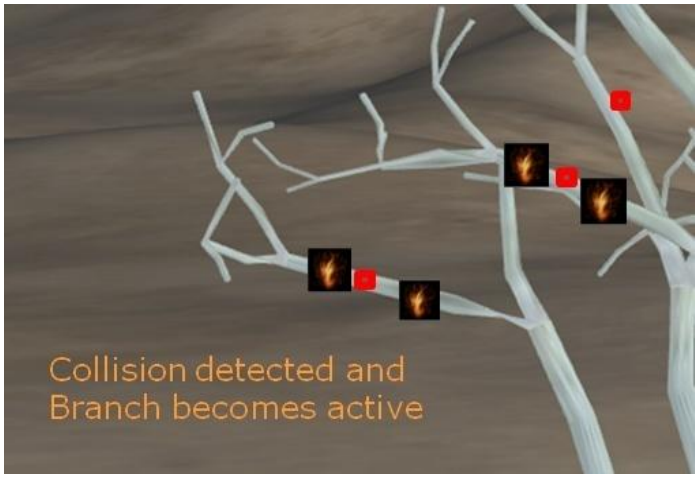
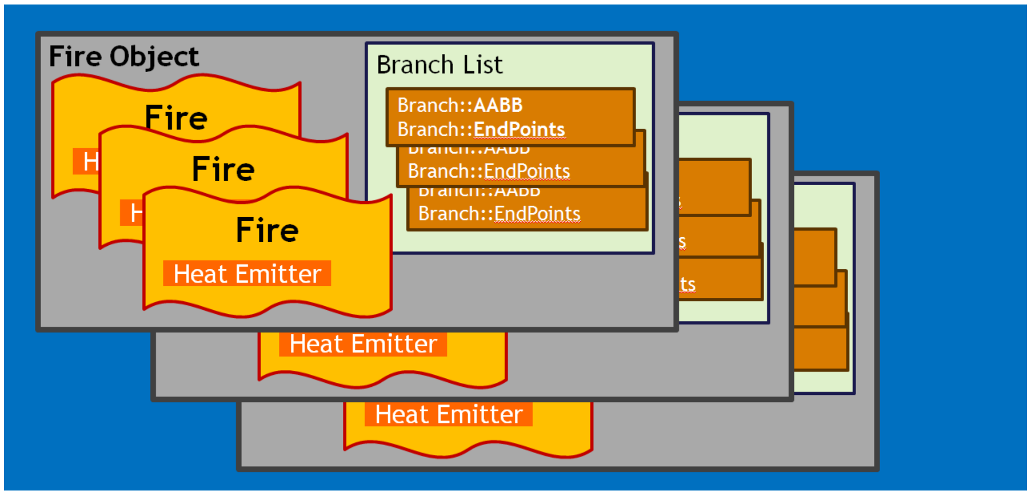


Figure 19 As More Heat Particles are Emitted the Fire Spreads

With multiple smart particles systems active at a single time, thermal mass is created. By extending the concept of thermal mass, you can use fire types other than just a line, such as a sphere, a box, or a patch.

This paper describes the Fire System, Fire Objects, and fires. The Fire System governs fire behavior. A Fire Object has one or more fires. A tree could be a Fire Object with fires for each branch. A meteor could be a Fire Object with a single Sphere Type fire. A house could be a Fire Object where each plank or shingle would be a separate fire.



6.1.1 Some Examples

This example uses a single tree without a canopy.

Non-Visible Heat Particles

Non-visible heat particles have a switch in the configuration that allows you to see the heat particles. In this case, shown in the figures below, the base of the trunk is on fire and the trunk is emitting heat particles.

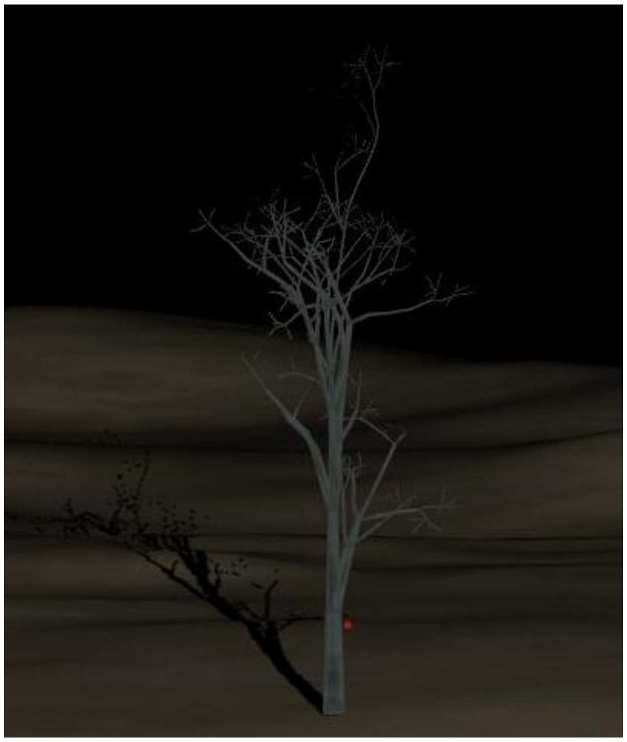


Figure 20 One Heat Particle (RED) Emitted From the Trunk

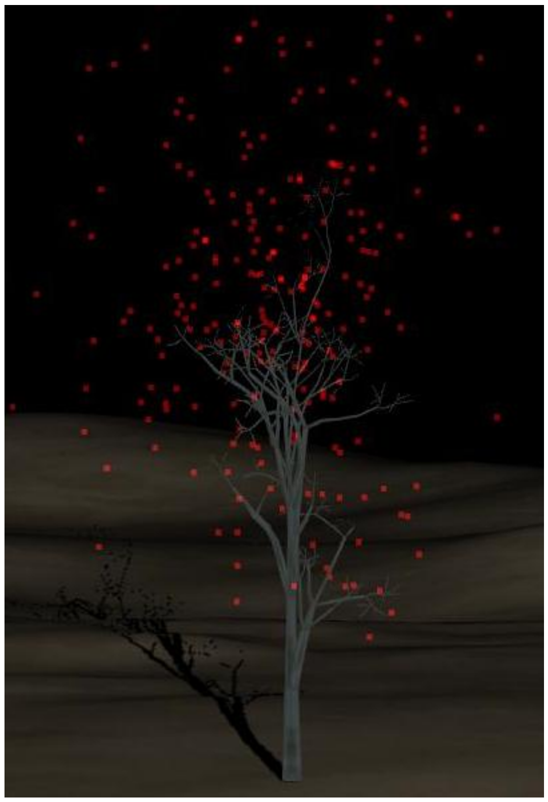
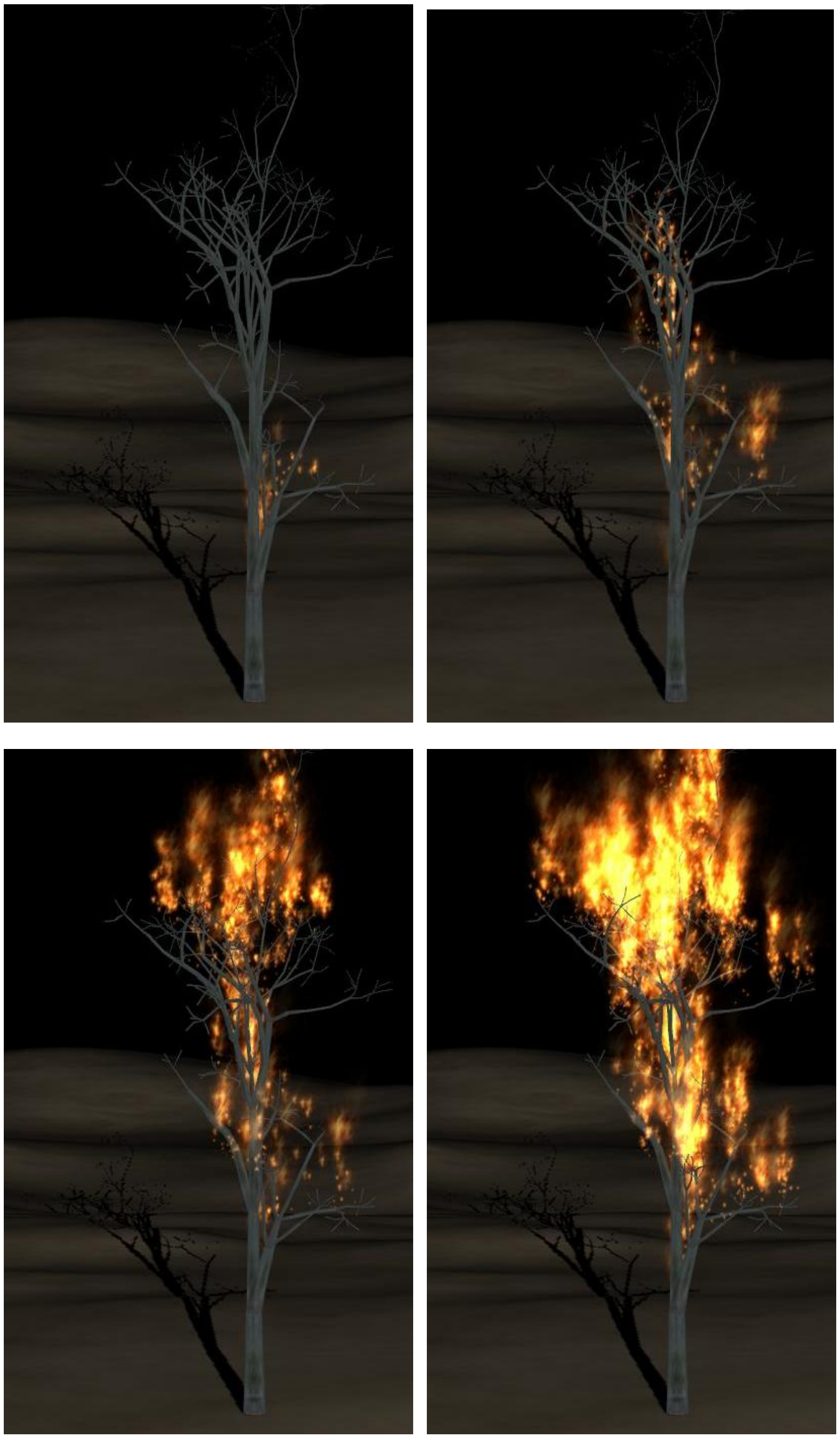


Figure 21 Tree Fully Engulfed Producing Many Heat Particles

Visible Particles



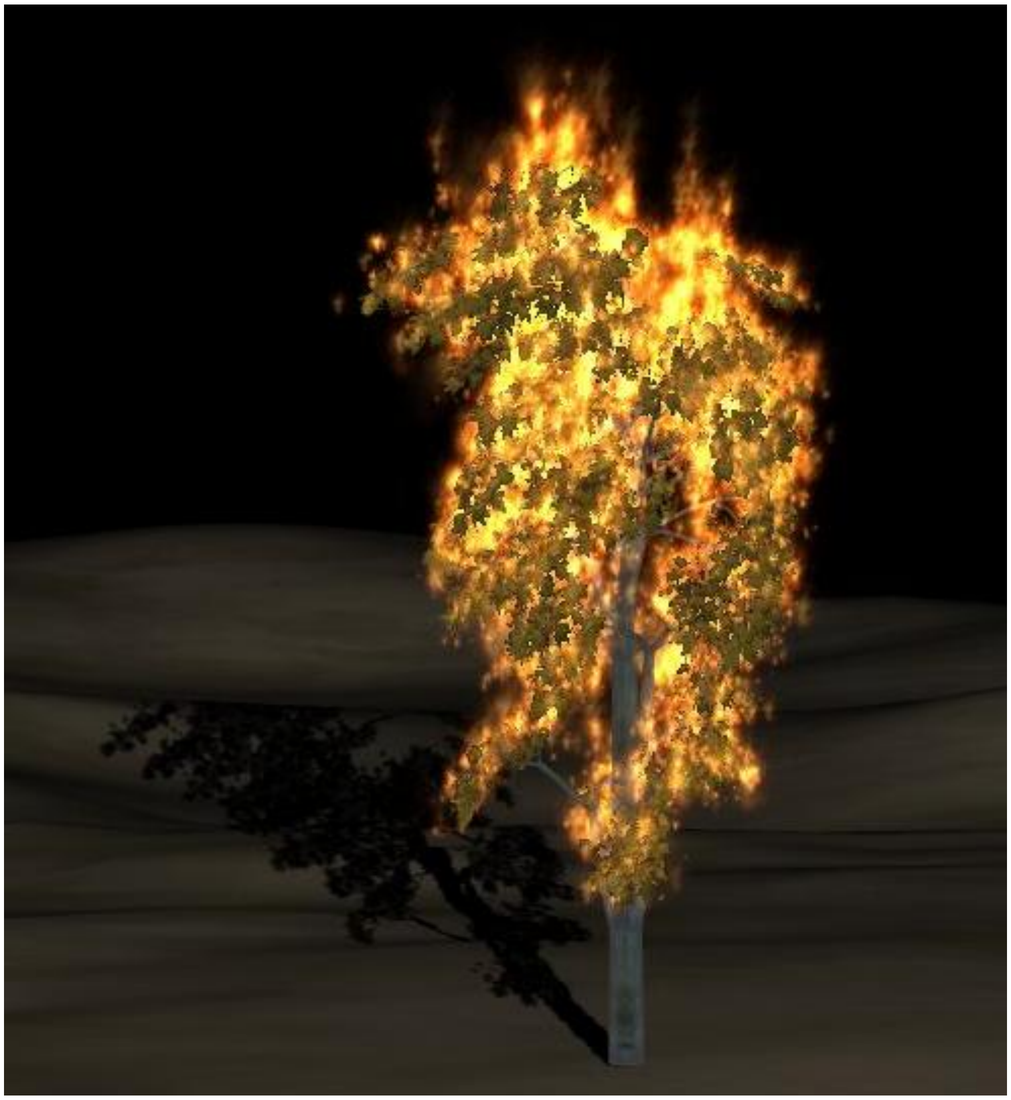


Figure 22 Engulfed Tree with the Canopy

6.2 Heat Particle Interaction

As discussed earlier, heat particles can interact with other fires. They can also act on any other fire in any other Fire Object. This means that when a heat particle from one fire intersects the bounding box of any other inactive fire you can make the fire spread. This is true not just within a single Fire Object but also from object to object. This behavior helps to make the effect realistic. It also limits any tedious scripting.

Imagine that there is a flaming meteor crashing to earth, and you want it to cause the trees to catch fire. Because the meteor is a Fire Object the fire will spread from the meteor to the tree. You can point the meteor at any tree and if the meteor is emitting heat it will set things in its proximity on fire. If you are working with a grove of trees, you can start the fire on any branch and the fire will spread based on the behavior of the heat particle parameters you specified: branch to branch, tree to tree, Fire Object to Fire Object.

6.3 Fire Interaction

To cause the fire to spread you must check the interaction of every active heat particle with every inactive fire every frame. Each fire has its own heat particles, each Fire Object has its own fires, and the Fire System has Fire Objects. Such scenarios are exactly what the Intel multi-core technologies are designed for.

Fire Spreading Algorithm

```
For each Fire Object
  For each fire in all Fire Objects
    For each heat emitter
      Check collision (All non burning branches)
        if collision
          set branch to burning state
```

The heat particles are maintained in local space to the Fire Object. This means before you can check the collision of the particle with the fire you need to make sure the Fire Object and fire are in the same space when the collision detection occurs.

Translate to a Common Space

For each heat particle

- Translate from parent object’s local space to world space.
- Translate the parent’s heat particle from world space to the subject’s local space.
- Check for a collision with the subject fire bounding box.

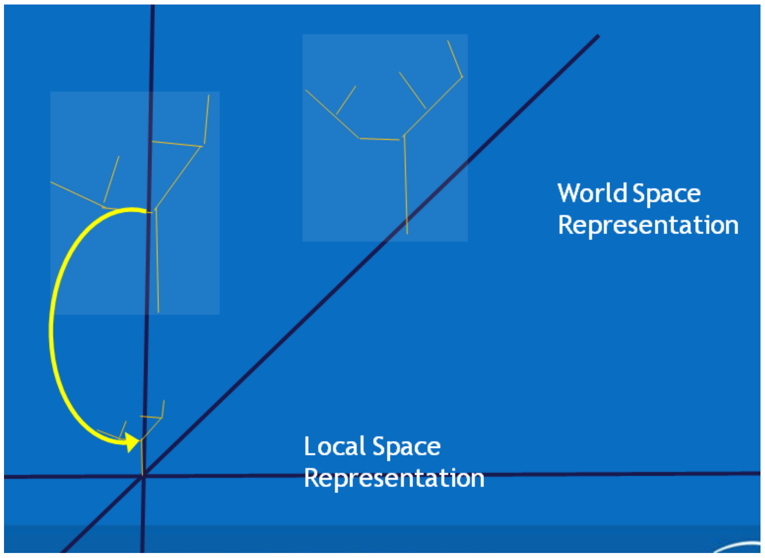


Figure 23 Each Tree Has its Own Local Space

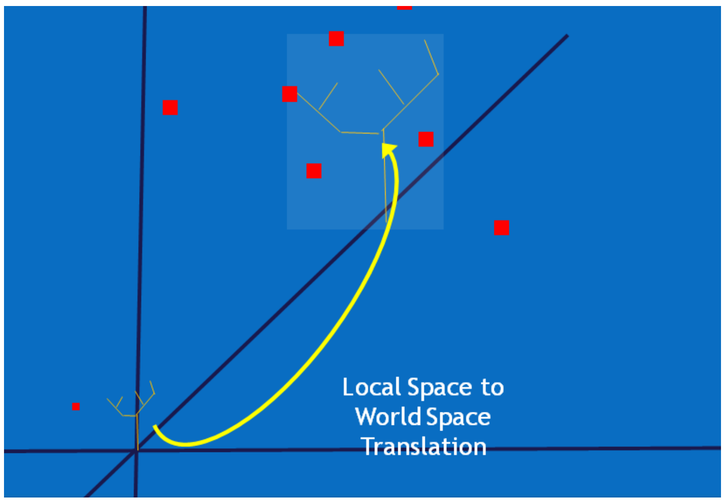


Figure 24 The Particle Gets Translated to Parent World Space

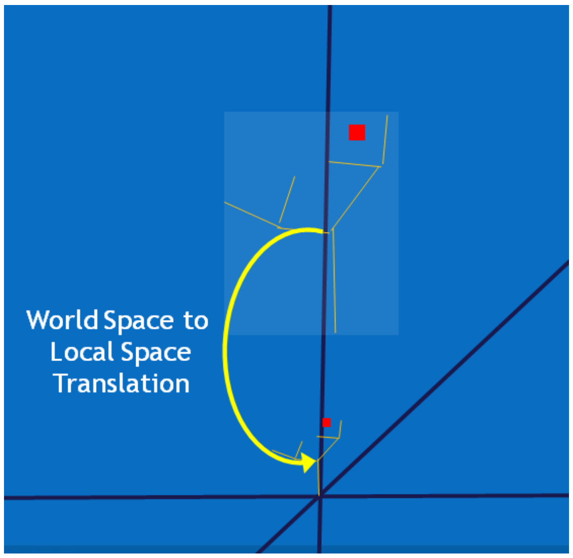


Figure 25 The Particle Then Goes From World Space to the Subject's Local Space

6.4 Heat Particle Behavior

Considerations for defining the parameters of a heat particle include its direction, its lifetime, and the number of particles to be generated by an emitter. Also, you need to take into consideration that fire tends to go more up than down or horizontally. Ideally, you could change them either statically or dynamically. For example, if there were wind blowing you might want the fire to spread in the direction of the wind.

Defining these characteristics requires familiarizing yourself with the concept of a velocity volume. The velocity volume gives a volume that defines the range of different vector values that can be produced for a given emitter. These velocity volumes are used for both the visible flame particles and the non visible heat particles. For each flame or heat particle there is some random vector it travels along for its lifetime. The random vector that is created for each particle must lie somewhere in the defined velocity volume. To define this range of possible trajectory vectors, first you must produce a vector based on a random XYZ basis. The outcome of this step is the volume of a cube of evenly positive and negative possibilities on every axis. To refine this velocity volume you can scale the vector along any axis to get a tighter volume. In the context of this document, this is "the impulse of the vector volume."

This process still only provides an even scaling along the positive and negative values of a given axis. In the case of this example, you need to have more positive values than negative on the Y axis. This is because heat rises, requiring a shift to be applied. Once your impulse and shift are defined you can apply these to any random vector contained in a unit volume. This gives you a trajectory vector for every particle that is bounded by the range of this velocity volume. The particle now has the desired predictable but random behavior.

Given an Impulse vector of (0.4, 1.0, 0.4) and a shift vector of (0.0, 0.8, 0.0), the following figures show the resulting vector volume.

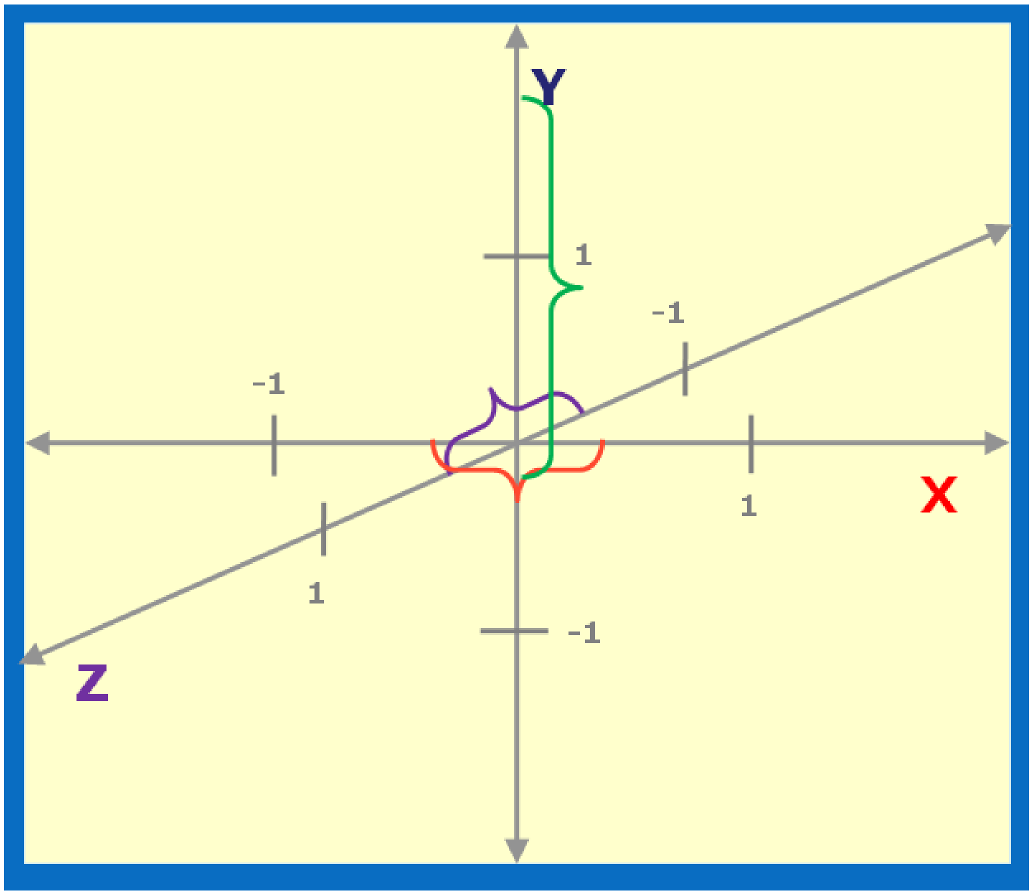


Figure 26 Impulse and Shift

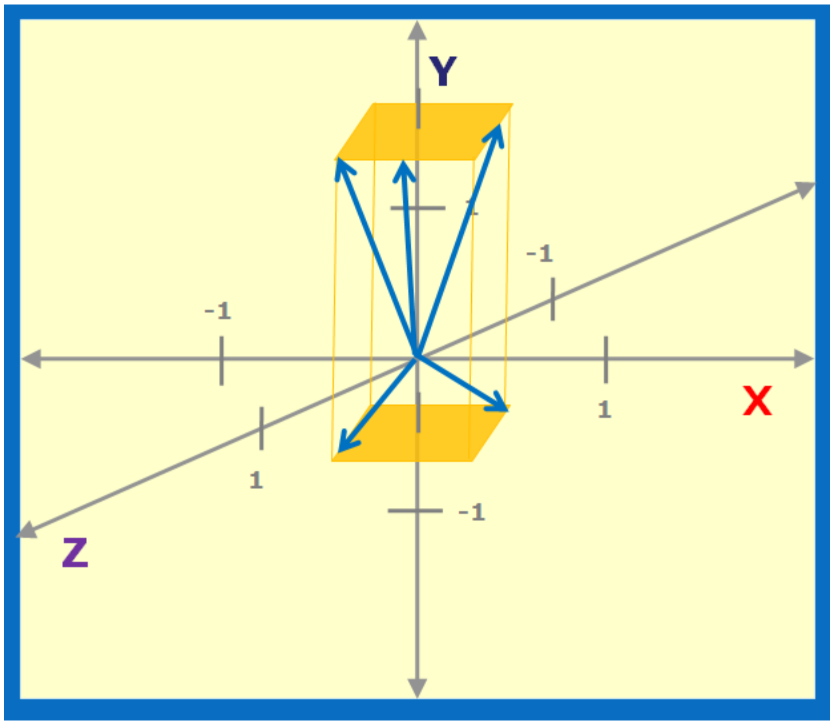


Figure 27 Velocity Volume

7 Future Considerations

Fire System Optimizations

The Fire System was a complex creation. Given the initial complexity and newness of this feature, much of the time spent on development was spent making sure the system behaved as expected. Time became short on fine tuning some of the performance optimizations. As you add more trees to the forest, more heat particle systems are added to the collision detection arena. There are interesting characteristics of the work load as the fire starts, as trees are added to the “on fire” bucket and eventually become part of the “fully consumed bucket”. The number of fires added to the system is non-linear. It is beyond the scope of this document to address the performance details, and while it isn’t clear how much more performance can be gleaned from a focused study of the culling and threading choices made for this version, it is clear there are a lot of opportunities for review to refine the performance of the this system.

Burn and Destroy

Right now the trees and other objects only catch on fire. They do not burn down to ashes and the fuel source is never ending. It was originally planned to have the trees burn down, have branches fall off, and have textures and techniques change with time. Branches should have a fuel volume. As noted previously the life and burning parameters for the branchBase structure would be used for this.

Tree Movement

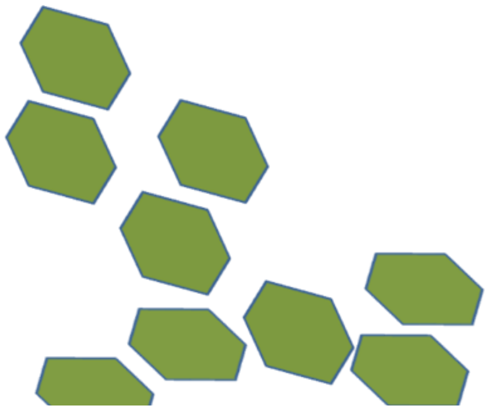
Since we have a full linked structure of each tree it would not be difficult to have the trees sway and move with wind or other impulse factors.

OOBB Fires

Presently the fire will move with the Fire Object. The fire and heat emitters follow the path of the meteors. With the trees it easy to create a patch for the fire for each canopy segment because you have knowledge of the tree as it is created. That is not the case for other objects that are not procedurally generated such as the house. For the fire to orient itself to the roof or other non orthogonal surfaces you need to be able to orient the patch fire to these surfaces. You also want the fire to follow a plank or shingle if it gets blown apart. This requires a Object Oriented Bounding Box (OOBB) implementation where you can quickly calculate the OOBB for each plank or shingle. The work on that was started but was not completed in time for the release of the Smoke project.

Hex Pattern Canopies

The parser for the canopies presently just creates a patch and then perturbs the patch vertices. It is kind of like crumpling a piece of paper and then slapping the canopy texture on it. It works very nicely for general effects. Another method that would be interesting to implement is, rather than a patch, to procedurally generate small hex patches that place themselves in pattern based on a grammar. This could produce more varied and realistic looking canopies.



Expanded Split Types

There are many expansions to the grammar that would be interesting, but of great value would be the ability to expand on the split types to easily create species to emulate pine trees or other symmetry based patterns.

Wind Silos and Wind Bubbles

Volumetric vectors for things like wind or smoke can be very intensive. The goal would be to focus on believable weather effects from wind. Imagine creating wind bubbles that would have an impulse vector. These bubbles could float around in a scripted or predicted path and as they intersected the velocity volumes of the particles they could dynamically change effects. Imagine also a bubble machine or machines that generated these bubbles. Wind silos would be similar in that you would have a cylinder that had a given number of vectors along its center. These would perturb and oscillate based on some weather noise and interact with the velocity volumes of the particles. Wind silos could be strategically placed to produce things like tornado or whirl wind effects.

Smart Particle Enhancements

There has been work that is included in the Smoke project on smart particles for a water system that puts out the fire. Other properties like gravity or temperature would be interesting to experiment with.

8 Conclusions

Procedural content in a virtual world is a wide open area of interest. The procedural trees and fire presented in this paper are just a small addition to the work that has been done.

The high compute intensity of the Fire System lends itself to a parallel implementation of the collision algorithms for all the particles and all the fires. It is a great example of the new types of features and effects that can be generated using the Intel multi-core technologies. The Smoke Framework illustrates how this was done for the Fire System.

If you are interested in the topics covered in this paper or would like to get the code for the Smoke Framework that has the procedural trees and fire implementation included then visit http://whatif.intel.com.

9 Appendix A Grammar and Parser Details

Example: XML Tree Species Grammar

```
<TreeGrammar>
  <GrammarElement Name="GDV">
    <Property Name="Name" value="TG1"/>
    <Property Name="LevelCount" value="6"/>
  </GrammarElement>
  <GrammarElement>
    <Properties GrammarToken="Level">
      <Property Name="LevelID" value1="0"/>
      <Property Name="tipPointCount" value1="13"/>
      <Property Name="AxialBias" MinAngle="30.0"/>
      <Property Name="AxialBias" MaxAngle="330.0"/>
      <Property Name="Length" value1="10.0"/>
      <Property Name="Diameter" value1="3.3"/>
      <Property Name="taper" value1="0.30"/>
      <Property Name="dropAngle" MinAngle="10.0"/>
      <Property Name="dropAngle" MaxAngle="25.0"/>
      <Property Name="dropAngle" BiasAngle="50.0"/>
      <Property Name="Heading" value1="0.0" value2="1.0" value3="0.0"/>
      <Property Name="Heading" MinAngle="-5.0"/>
      <Property Name="Heading" MaxAngle="5.0"/>
      <Property Name="branchType" value1="TRUNK"/>
      <Property Name="canopyType" value1="NULL_CANOPY"/>
      <Property Name="splitListcount" value1="4"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="0"/>
      <Property Name="probability" value1="0.25"/>
      <Property Name="splitCount" value1="5"/>
      <Property Name="type" value1="ORDINARY"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="1"/>
      <Property Name="probability" value1="0.05"/>
      <Property Name="splitCount" value1="2"/>
      <Property Name="type" value1="ORDINARY"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="2"/>
      <Property Name="probability" value1="0.15"/>
      <Property Name="splitCount" value1="3"/>
      <Property Name="type" value1="REPEAT_OPPOSED"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="3"/>
      <Property Name="probability" value1="0.55"/>
      <Property Name="splitCount" value1="2"/>
      <Property Name="type" value1="ORDINARY"/>
    </Properties>
  </GrammarElement>
  <GrammarElement>
    <Properties GrammarToken="Level">
      <Property Name="LevelID" value1="1"/>
      <Property Name="tipPointCount" value1="11"/>
      <Property Name="AxialBias" MinAngle="30.0"/>
      <Property Name="AxialBias" MaxAngle="330.0"/>
      <Property Name="Length" value1="7.0"/>
      <Property Name="Diameter" value1="2.0"/>
      <Property Name="taper" value1="0.20"/>
      <Property Name="dropAngle" MinAngle="35.0"/>
      <Property Name="dropAngle" MaxAngle="50.0"/>
      <Property Name="dropAngle" BiasAngle="40.0"/>
      <Property Name="Heading" MinAngle="-20.0"/>
      <Property Name="Heading" MaxAngle="20.0"/>
      <Property Name="branchType" value1="BRANCH"/>
      <Property Name="canopyType" value1="NULL_CANOPY"/>
      <Property Name="splitListcount" value1="5"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="0"/>
      <Property Name="probability" value1="0.05"/>
      <Property Name="splitCount" value1="3"/>
      <Property Name="type" value1="REPEAT_OPPOSED"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="1"/>
      <Property Name="probability" value1="0.05"/>
      <Property Name="splitCount" value1="3"/>
      <Property Name="type" value1="REPEAT_OPPOSED"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="2"/>
      <Property Name="probability" value1="0.25"/>
      <Property Name="splitCount" value1="3"/>
      <Property Name="type" value1="REPEAT_OPPOSED"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="3"/>
      <Property Name="probability" value1="0.45"/>
      <Property Name="splitCount" value1="2"/>
      <Property Name="type" value1="ORDINARY"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="4"/>
      <Property Name="probability" value1="0.20"/>
      <Property Name="splitCount" value1="2"/>
      <Property Name="type" value1="ORDINARY"/>
    </Properties>
  </GrammarElement>
  <GrammarElement>
    <Properties GrammarToken="Level">
      <Property Name="LevelID" value1="2"/>
      <Property Name="tipPointCount" value1="9"/>
      <Property Name="AxialBias" MinAngle="30.0"/>
```

```
      <Property Name="AxialBias" MaxAngle="330.0"/>
      <Property Name="Length" value1="6.0"/>
      <Property Name="Diameter" value1="1.0"/>
      <Property Name="taper" value1="0.33"/>
      <Property Name="dropAngle" MinAngle="35.0"/>
      <Property Name="dropAngle" MaxAngle="50.0"/>
      <Property Name="dropAngle" BiasAngle="40.0"/>
      <Property Name="Heading" MinAngle="-20.0"/>
      <Property Name="Heading" MaxAngle="20.0"/>
      <Property Name="branchType" value1="BRANCH"/>
      <Property Name="canopyType" value1="NULL_CANOPY"/>
      <Property Name="splitListcount" value1="3"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="0"/>
      <Property Name="probability" value1="0.25"/>
      <Property Name="splitCount" value1="1"/>
      <Property Name="type" value1="REPEAT_OPPOSED"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="1"/>
      <Property Name="probability" value1="0.25"/>
      <Property Name="splitCount" value1="1"/>
      <Property Name="type" value1="OPPOSED"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="2"/>
      <Property Name="probability" value1="0.5"/>
      <Property Name="splitCount" value1="2"/>
      <Property Name="type" value1="ORDINARY"/>
    </Properties>
  </GrammarElement>
  <GrammarElement>
    <Properties GrammarToken="Level">
      <Property Name="LevelID" value1="3"/>
      <Property Name="tipPointCount" value1="7"/>
      <Property Name="AxialBias" MinAngle="30.0"/>
      <Property Name="AxialBias" MaxAngle="330.0"/>
      <Property Name="Length" value1="4.0"/>
      <Property Name="Diameter" value1="1.0"/>
      <Property Name="taper" value1="0.50"/>
      <Property Name="dropAngle" MinAngle="55.0"/>
      <Property Name="dropAngle" MaxAngle="70.0"/>
      <Property Name="dropAngle" BiasAngle="60.0"/>
      <Property Name="Heading" MinAngle="-20.0"/>
      <Property Name="Heading" MaxAngle="20.0"/>
      <Property Name="branchType" value1="BRANCH"/>
      <Property Name="canopyType" value1="NULL_CANOPY"/>
      <Property Name="splitListcount" value1="2"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="0"/>
      <Property Name="probability" value1="0.10"/>
      <Property Name="splitCount" value1="1"/>
      <Property Name="type" value1="REPEAT_CANOPY"/>
    </Properties>
    <Properties GrammarToken="SplitList">
```

```
      <Property Name="splitID" value1="1"/>
      <Property Name="probability" value1="0.9f"/>
      <Property Name="splitCount" value1="4"/>
      <Property Name="type" value1="OPPOSED"/>
    </Properties>
  </GrammarElement>
  <GrammarElement>
    <Properties GrammarToken="Level">
      <Property Name="LevelID" value1="4"/>
      <Property Name="tipPointCount" value1="7"/>
      <Property Name="AxialBias" MinAngle="30.0"/>
      <Property Name="AxialBias" MaxAngle="330.0"/>
      <Property Name="Length" value1="3.0"/>
      <Property Name="Diameter" value1="0.5"/>
      <Property Name="taper" value1="0.0"/>
      <Property Name="dropAngle" MinAngle="70.0"/>
      <Property Name="dropAngle" MaxAngle="90.0"/>
      <Property Name="dropAngle" BiasAngle="75.0"/>
      <Property Name="Heading" MinAngle="-40.0"/>
      <Property Name="Heading" MaxAngle="40.0"/>
      <Property Name="branchType" value1="STEM"/>
      <Property Name="canopyType" value1="NULL_CANOPY"/>
      <Property Name="splitListcount" value1="2"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="0"/>
      <Property Name="probability" value1=""/>0.0
      <Property Name="splitCount" value1="0"/>
      <Property Name="type" value1="ORDINARY"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="1"/>
      <Property Name="probability" value1="1.0"/>
      <Property Name="splitCount" value1="0"/>
      <Property Name="type" value1="ORDINARY"/>
    </Properties>
  </GrammarElement>
  <GrammarElement>
    <Properties GrammarToken="Level">
      <Property Name="LevelID" value1="5"/>
      <Property Name="PerturbFactor" value1="2.0"/>
      <Property Name="branchType" value1="NULL_BRANCH"/>
      <Property Name="canopyType" value1="PATCH"/>
      <Property Name="splitListcount" value1="1"/>
    </Properties>
    <Properties GrammarToken="SplitList">
      <Property Name="splitID" value1="0"/>
      <Property Name="probability" value1="1.0"/>
      <Property Name="splitCount" value1="0"/>
      <Property Name="type" value1="ORDINARY"/>
    </Properties>
  </GrammarElement>
</TreeGrammar>
```

10 References and Related Articles

[1] The Algorithmic Beauty of Plants by Przemyslaw Prusinkiewicz and Aristid Lindenmayer

[2] Real Time Design and Animation of Fractal Plants and Trees by Peter E. Oppenheimer New York Institute of Technology Computer Graphics Lab

[3] Creation and Rendering of Realistic Trees by Jason Weber Teletronics International, Inc. Joseph Penn Army Research Laboratory

[4] Frank Luna. Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach. Wordware Publishing Inc.

[5] Gamefest presentation on the same topic: http://www.microsoft.com/downloads/details.aspx?FamilyId=8CE4DA8F-E09C-4E40-A321-DFD0822E31A6&displaylang=en

Or http://www.xnagamefest.com/presentations08.htm

11 About the Author

Hugh Smith is a Sr. Software Engineer working on client enabling in the Software Services Group that enables client platforms through software optimizations. His passion is graphics and gaming. His e-mail is hugh.a.smith@intel.com.