Aspect-Oriented Procedural Content Engineering for Game Design

Walter Cazzola
DICo, Università di Milano, Italy
cazzola@dico.unimi.it

Diego Colombo
Microsoft Ireland Research, Ireland, and IMT Lucca, Italy
colombod@di.unipi.it

Duncan Harrison
Realtime Worlds, Uk
duncan.harrison@realtimeworlds.com

ABSTRACT

Generally progressive *procedural content* in the context of 3D scene rendering is expressed as recursive functions where a finer *level of detail* gets computed on demand. Typical examples of content procedurally generated are fractal images and noise textures. Unfortunately, not always the content can be expressed in this way, developers and content creators need the data to have some peculiarity (like windows on a wall for a house 3D model) and a method to drive data simplification without losing relevant details.

In this paper we discuss how *aspect oriented* (AO) techniques can be used to drive the content creation process by mapping each data peculiarity to the code to generate it. Using *aspects* will let us to partially evaluate the code of the procedure improving the performance without losing the flow of the generation logic. We will also discuss how the use of AO can provide techniques to build simplified version of the data through code transformations.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Aspect-Oriented Programming

Keywords

AO Techniques, Data Engineering, Meta-Data

1. INTRODUCTION

In a game application most of the content is composed of 3D models and other geometric objects used to populate the screen. In a disc based scenario the game data is usually compiled into a very concise binary format, this approach when applied to consoles means that we can have a single continuous block of data on the storage. Such a data representation affords the ability to load a large sequential chunk of bytes from the storage medium and have all of the information ready to be used by the application code [4]. Making a concise representation of the data is crucial for this scenario so that the application can start presenting coarse representations of content while waiting for the fine grain data to load and be prepared for presentation (different level of detail in different moments).

To use this method coarser representations of the game content are built providing a faster start up and affording memory usage optimization at run-time. Offsetting this is the cost of extra space on the storage to save simplified data along with the high quality content, and a smart lookup strategy. Bandwidth is a factor that is quite important when a next generation game is loading live from a DVD. Smart caching techniques can be used to address this problem but are applicable only on specific hardware, another way to reduce bandwidth is to generate the data on-the-fly and to store/-transmit only the function and the inputs to produce them. Instead of storing a cube in an explicit manner, usually model data is stored as a set of faces and vertices, we store the geometric primitive as a position and set of parameters that can be used to generate the unique instance. This provides a potentially significant saving over the amount of storage space required for the explicit alternative. Computational cycles are far less expensive than content delivery bandwidth, a procedural generation from a very concise representation can lead to a drastic requirement relaxation for streaming based applications. Procedural generation of content (and we will refer to this technique as *procedural content* generation) is a set of input values and the code to build the destination data. Given $\mathcal{G}$ the set of *generator functions* the procedural content is defined as:

$$\bigcup_{g \in \mathcal{G}} \{g(s) \mid \forall s \in \text{Domain}(g)\}$$

in a specific point the procedural content is represented by the couple (g,s) where $g \in \mathcal{G}$ and $s \in \text{Domain}(g)$.

In our case, the chosen family of generators must output data displayable on screen. So the co-domain of the generators is the set of the, so called, *renderable mesh*) defined as:

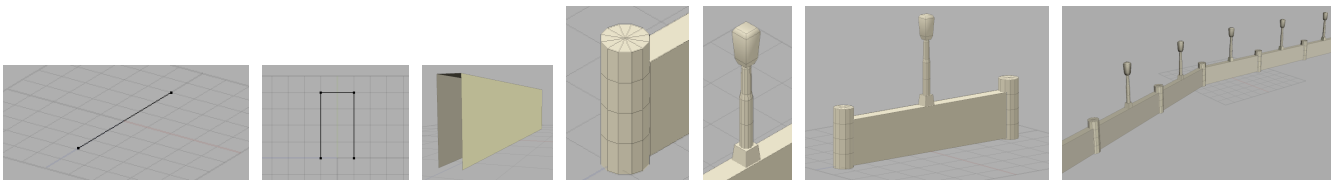
$$RM = \{(m,g) : m \text{ is a graphics material, } g \text{ is the geometry}\}$$

and the generator, called *inflator*, is represented by

$$\text{inflator}_i : \text{Domain}(\text{inflator}_i) \rightarrow RM$$

Designing and engineering procedural content is quite important because the amount of data can increase (especially from a semantic perspective) whilst keeping the occupied space constant or at least bounded. A lot of *digital content authoring tools* (we will refer them as DCC) store user files in a way that the content affords easy manipulation instead of an optimised raw data set. Quite often data are kept as extensive as possible in the content production pipeline and get finalised once built for the final application. This

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.



(a) polyline segment (b) profile to be extruded (c) extruded polygon (d) cylindrical pole (e) lamppost (f) wall section (g) full wall

Figure 1: From a polyline to the final picture.

finalisation process usually entails the generation of coarser granularity *levels of detail* and the computation of final layout on storage. This can be thought of as storing data as a program with an input set until ready for the final build where the target product will use the program's output with the given input set. Most of the strategies for content simplification deal with the final content and not with the content generation procedure where the static output is needed to obtain coarser version of the data. Changing the perspective from data to program can lead to a different paradigm for content engineering, more related to the data synthesis than to the data representation itself. The video game Crackdown[1] from Realtime Worlds[2] uses procedural built cities and road, this means that, on the disc, data are represented by the code to generate the content and the input data for the procedure and the rendering step.

Let us consider a wall generator that takes as input a polyline, defining the way the wall geometry should be generated from the path we can specify it by listing peculiarities of the final object we want to obtain. So we say that the body is the extrusion of a path along the wall direction, poles are distributed along the wall at every point and lampposts are laid out along the wall on the top surface every 10 meters. Any peculiarity of the final mesh can be represented as aspect in the aspect-oriented paradigm and therefore we can obtain the geometry builder for that style of wall by using a polyline as input and *just* weaving the right collection of aspects.

The aim of this article is to propose the usage of modern modularization techniques like AOP [6] to design and simplify content acting on the behavioural aspect of the procedure instead of the output obtained from it; AO techniques with run-time weaving can reduce the storage and bandwidth requirements for data and take advantage of hardware acceleration where available. The use of procedurally generated content has a measurable impact on system memory requirements: the meshes rendered on the screen are stored in the video memory loading a mesh from disc will locate it in the system memory, then the hardware friendly version of each vertex is built and sent to the video memory for rendering. When a physics engine is involved there is usually an additional mesh optimized for physic simulations. Building the mesh procedurally means that there is no need for the original copy to be totally loaded in the system memory, graphics and physics data can be built at runtime reducing the pressure on the memory management system.

2. COARSER LEVEL OF DETAILS

Building coarser data representations is mandatory for game design, coarse data are required to hide fine data loading time whilst maintaining a level of visual coherency and to bound memory usage. In this section some of the most common techniques are illustrated. Let LOD_0 be the highest detailed representation of a data and LOD_n the reduced representation at stage n. We can state that:

$$\{LOD_i\}_{i\in\mathbb{N}} \subset RM$$

and from the set of *generator* functions we will have:

$$\text{LODGen}_i : RM \to RM$$

where the subscript identifies which *LOD* will be generated.

Data reduction. With a static data representation it is possible to build less complex representations of the original data, minimizing the impact due to the loss of quality. On a 3D dataset it is quite common to use polygon reduction (*decimation* [8]) removing details deemed irrelevant given the context (distance from the camera, lighting condition). Occasionally polygon reduction is integrated with redetailing strategies [3] to hide the loss of information. When reduction occurs an error is introduced, since candidates are selected for removal to minimise the error, common criteria include removing faces beneath a surface area, edges shorter than a given length, etc. Combining more criteria is generally a good strategy to preserve the relevant characteristic of a 3D dataset.

Progressive meshes. When content is produced by a procedure a distinction is made between continuous and discrete *LOD*s. Progressive meshes [5] are an example of continuous *LOD*. A progressive mesh (*pmesh*) is built from a mesh and the output is a data structure that can change level of detail based on a parameter. If the input mesh is altered the pmesh must be rebuilt. Some techniques must be applied to preserve the coherence of the coarser versions of the pmesh but their description is out of the scope of this paper.

Grammar based content. As shown in [7] a grammar can be used to express geometries. The main concept is to provide primitive geometric figures like cubes, boxes, and cylinders which are applied based upon grammatical rules to a symbolic string representation. Every terminal symbol in the grammar is mapped to a polygon, so we can have `A` symbol to be associated with a rounded window and `B` to a squared one; so `(ABA)+` will be expanded to a sequence with at least one pattern of rounded-squared-rounded windows. By applying this approach to our example we will have the following productions for the wall generation:

```
Wall        :  (WallSegment)+
WallSegment :  (Pole WallBody (WallSegment)* Pole)+
WallBody    :  Rectangle Lamppost
```

To get automatic *LOD* generation some extra work is needed. To preserve the grammar based approach instead of applying the data reduction approach we need to build another grammar where the tokens will produce simplified version of each model peculiarity and a rewriting rule to transform the original to the new grammar.

Data reduction approaches require fully generated content as input to a simplification process. This requirement is counter intuitive as at run-time the coarse data are intended to hide the cost of generating high detail data. The pmesh approach is based on data preprocessing: the high quality mesh is compiled into a dataset (the

[1] Crackdown at `http://crackdownoncrime.com`.
[2] Realtime Worlds at `http://www.realtimeworlds.com`.

```
public class ProceduralWall : ProceduralContent {

  // the polyline describing the wall
  public List<Point3Df> PolyLine { get; set; }

  // Constructor
  public ProceduralWall() {
    PolyLine = new List<Point3Df>();
    // Load the data for the Polyline
                ...
  }

  public override RenderableMesh Inflate() {
    BuildMesh(Polyline, out m_RenderableMesh);
    return base.Inflate();
  }

  public void Build(IEnumerable<Point3Df> _path,
                ref RenderableMesh _rm) {

    // Original Code
    return;
  }
}
```

```
// The profile to be extruded
public List<Point3Df> Profile {get; set;}
```

```
// Load the data for the Profile
Profile = new List<Point3Df>();
...
```

```
// Code for extruding the profile along the path
...
```

Listing 1: `ProceduralWall` through a 2-stages weaving

pmesh) which is efficiently reducible. Any alteration of the original mesh necessitates pmesh regeneration, this approach is therefore not a good choice for highly mutable scenarios such as games. Grammar based systems introduce additional overheads which are undesirable in context, simplification takes place through a rewriting system that replaces productions in the grammar and implies a *continuous* data computation.

3. DATA ENGINEERING WITH AOP

Aspect-oriented [6] techniques applied to the procedural content generation helps in overcoming the abovementioned issues. Our examples are developed by applying the AspectWerkz aspect-oriented approach [9] to C#: aspects are purely C# classes with XML driven fragment weaving. In our AO approach the aspects are dynamically assembled and woven to compose the generator. This permits us to choose the code to weave after some dynamic criteria (performance, available bandwidth, ...). The DCC tool is used to generate a description of the aspect. The application (the game in our scenario) will use such a description and some ancillary code to assemble the aspects; then it proceeds to the weaving step. Placing the focus on the code perspective for procedural content we can express a procedural *LOD* generator as:

$$\text{ProcLODGen}_i : \mathbb{INFL} \rightarrow \mathbb{INFL}$$

where $\mathbb{INFL} = \{\text{inflator} \mid \text{inflator} : RM \rightarrow RM\}$ and the subscript identifies the *LOD* returned by the selected inflator.

Using the ProcLODGen_i function rather than the LODGen_i permits the creation of *LOD* as a simplified version of the aspect. This is computed by the application without the need to compute intermediary simplified versions of the original data and without multiple aspect definitions. To simplify the aspect description a set of rules will be provided to the aspect generator.

3.1 Building Mesh Builders

Let us consider our wall generator example, we can define the base class `ProceduralContent` as:

```
public class ProceduralContent {

  // the data structure for the rendering process
  protected RenderableMesh m_RenderableMesh;

  /* the Renderable data structure is initliased
     before the Inflate method get invoked */
  public virtual void CreateRenderableMesh() {
    m_RenderableMesh = null;
  }

  // the RenderableMesh is built and returned
  public virtual RenderableMesh Inflate() {
    return m_RenderableMesh;
  }
}
```

The `m_RenderableMesh` value, our procedural content, is not computed by this class but demanded to the `inflate()` method of the `ProceduralWall` class (Listing 1 neglecting the code in the arrowed boxes). As previously described the content generator for the renderable mesh depends on the desired *LOD* and cannot be hard-coded in the class because this would imply that the function would require code to support any possible *LOD* thereby hampering the possibility of augmentation or reduction of the *LOD*.

A group of aspects take care of the different *LOD*s and the corresponding mesh content. In respect to this, we can have an extensible library of geometry generators that can be used to build the aspects introducing the necessary code to generate the procedural content. The class for the extrusion needed to model the example is defined as follows:

```
/* perform the extrusion of a Profile along a
   path; the profile is advised as well */
public class ProfileExtrusion {

  // The profile to be extruded
  public List<Point3Df> Profile { get; set; }

  /* Extrude the profile along the given path; the
     renderable mesh is modified in place */
  public void Extrude(IEnumerable<Point3Df> _path,
              ref RenderableMesh _rm) {

    /* For each segment in _path appends the Pro-
       file extrusion along the segment to _rm */
            ...
  }
}
```

The class must build the 3D wall in Fig. 1(c) as an extrusion of the 2D profile in Fig. 1(b). The body of the method `Extrude()` must be woven into the `ProceduralWall` type before the execution of the method `Build()` to correctly generate the target media. The weaver must also build the bindings to the open variables in the code extruded from the method `Extrude()` and set-up the geometry of the `ProfileExtrusion.Profile` field.

Pointcuts cannot be defined along with the `ProfileExtrusion` because the aspects themselves are not defined yet, shown classes are part of the framework but the aspects and the pointcut will be defined later by the DCC. The weaving process will take place when the request for content is raised. Even if the weaving process is just a (customized) intermediate code rewriting, we will refer to the rewriting system as a weaver.

The weaving process must be multi-stage, modelling the intrinsically incremental nature of the procedural content. For example, to put a pole 10cm higher than the wall at every point of the polyline we must know the wall height, information available only after a partial evaluation of the procedural content. Considering our scenario, in the first stage, we introduce into the `ProceduralWall` class the necessary code to describe the profile to be extruded (Listing 1, injection fragment set 1). In the second stage, we add the code to initialize the `Profile` property of the class by further extending the constructor (Listing 1, injection fragment set 2). For sake of clarity, Listing 1 shows the C# code we should get if the weaving process will be done on the source rather than on the intermediate code. As you can see, the procedural content is generated as the result of the incremental weaving of new algorithms to those already contributing to the computation; the new algorithms can use partially evaluated data and adapt consequently. Moreover, each stage contributes to the preparation of the next increment by building the pointcut expressions, based on the previously computed content, used to locate the code to add and where to add it.

Referring to figure 1(f) the decorating elements of the wall such as poles and lampposts are realised by different classes. Behaving as decorators dictates that their position is dependent on the `PolyLine` property of the `ProceduralWall` class and on the previously computed wall size. To apply decorations to the wall it is necessary to inject their initialisation code after the code of the original constructor and solve the dependencies to weave correctly. In our example the lamppost and light attachments require knowledge of the segment centre point and the location of the walls upper surface respectively. These variables must be computed in the preceding stage and forwarded through fragment dependencies.

The advantage of our AO approach when applied to engineering *procedural content* is that elements such as `ProceduralWall` and `ProfileExtrusion` are defined in the application framework while the pointcuts will be produced by the content creators in DCCs allowing emission of bytecode in place of plain data structures. DCCs will not perform weaving because it can be done on the client affording a higher degree of flexibility, we expect editors to build the sequence of stages the content will be transformed by and the pointcuts for each stage.

The XML in Listing 2 is the output of the DCC elaboration and it will be used on the client to carry out the weaving process by using each `<Stage>` node to define *pointcuts* and *advices*. The `ExtrudeMethod()` node is a child of the `<Source>` node, all the children of this node are marked for the code extrusion; the `<ParameterMapping>` element provides bindings to the open variables. In this example once the code for the `Extrude()` has been extruded from the `ProfileExtrusion` class we will change every reference to the arguments with the arguments of the target method (`BuildMesh()` in the `ProceduralWall` class).

3.2 Mapping Aspects on Annotations

The code to manage the XML in Listing 2 would bloat even for such a small example. It has to describe how to build the poles at every point `p` of the polyline defining the path of the wall. Similarly, it must expand the code to the necessary level to make a cylinder at `p` with given dimensions and so on for any decorating element. In the XML code, we direct the weaver to inject the code through *around/before execution* statements; to speed-up the generator execution the code should be inlined.

Extrude-Inject-Evaluate is the strategy used to generate the final `ProceduraWall` class, reducing the number of calls to external methods and boosting performance. To this end *multi-resolutions joint points* are necessary, i.e., they must be able to address a method

```
<Node type="ProceduralWall">
 <Stage>
  <Introduction mode="Property" modifier="public">
   <Property name="Profile" type="List<Point3Df>"/>
  </Introduction>
  <Before>
   <Execution methodName="BuildMesh">
    <Source>
     <ExtrudeMethod method="ProfileExtrusion.Extrude">
      <ParameterMapping name="_path" scope="args">
       <Var name="_path" scope="args"/>
      </ParameterMapping>
      <ParameterMapping name="_rm" scope="args">
       <Var name="_rm" scope="args"/>
      </ParameterMapping>
     </ExtrudeMethod>
    </Source>
   </Execution>
  </Before>
 </Stage>
 <Stage>
  <Around original="before">
   <Execution methodName=".ctor">
    <ExtrudeMethod method="DataLoader.LoadProfile">
     <ParameterMapping name="_profile" scope="args">
      <Var name="Profile" scope="members"/>
     </ParameterMapping>
    </ExtrudeMethod>
   </Execution>
  </Around>
 </Stage>
</Node>
```

Listing 2: Data from the DCC

call, the body of a method and onwards down to a single instruction. In our proposal, we adopt [a]C# [1], whose annotation model permits us to annotate declarations and statements as well, to annotate the portion of code that represents a join point for code weaving. The particularity of our join point model (quite similar to that of AspectJ 5+[3] and AspectWerkz[4]) consists of explicitly marking the join point of interest through annotations.

In Sect. 3.1, we stated that we need to use the woven code at stage n to build the pointcuts for stage $n+1$, this will not allow code inlining since we cannot create the new pointcuts dynamically. Annotations can help us in achieving both code inlining and pointcut dynamic generation because we can keep track of the woven/inlined code through the *annotation scope* associated to the code fragment. A code fragment can be denoted by an annotation [1] and refer to the annotation itself, the annotated bytecode and its closure, the annotation scope. Let us consider the following `ProceduralWall` constructor with an empty *fragment*.

```
public ProceduralWall(){
  [LoadData(Target = Polyline)] {}
}
```

`LoadData` is the fragment type and the `Target` property of the annotation is set to the `Polyline` property of the `ProceduralWall` class. Additionally in our approach enclosing methods and the type are part of the annotation scope. The full annotation hierarchy in the example is:

```
ClassScope(ProceduralWall)⊂ ConsScope⊂
  LoadData(Target = "this.Polyline")
```

[3] `www.eclipse.org/aspectj/`
[4] `aspectwerkz.codehaus.org`

where the «Elem»Scope(X) represents the annotation scope for an annotation decorating the corresponding «Elem»[5] named X. These annotations are implicitly derived from the types themselves, they are not explicitly visible in the code. Note that code fragments can be both nested and in sequence [1].

A small set of operations are defined on the annotation scopes. They permit us to: *retrieve* the code fragment, *inject* a code fragment before or after the start or the end of the annotation scope, *extrude* a fragment and its closure and to *delete* the code fragment.

Code fragments are used as source for both the aspect generation and joint point in the weaving process. The LoadData fragment is empty as the polyline is normally serialised along with the class instance but it can be refined during the execution. For example, the code (CollapsePolylineSegments(List<Point3Df> path) method) to collapse together the segments shorter than a given length can be added to the LoadData fragment as a new annotation scope.

```
[CollapsePolylineSegments]{...}
```

This method directly acts on the path argument in place and inlines the CollapsePolylineSegments fragment in the LoadData fragment then binding the path argument to the Polyline property. The binding will replace the Ldarg_1 with the IL to get the property Polyline. The final annotation skeleton we obtain is:

```
[ClassScope(ProceduralWall)]{
  [ConsScope()]{
    [LoadData(Target = "this.Polyline")] {
      [CollapsePolylineSegments] {...}
    }
  }
}
```

Annotations can also be used to describe fields and properties to accomplish with introductions. Moreover, it is possible to *delete* operations marked by annotations; this is a feature unsupported by most of the AO tools. Being able to delete an annotation and its closure is a key operation for performance boosting. Care must be taken however to only remove members and variables unique to the fragment, that is those which are not in any other fragment's closure. .NET has a rule for method inlining[6] that is based on the method size; thus reducing the method size by removing all the "*undesired*" code will increase the chance to get the method call inlined by the JIT. An additional benefit resultant from stripping out unused code is reduction in memory footprint.

Now we can express the aspect definition as:

```
public class GeneratedAspect {
  public GeneratedAspect(){
    Inject(
      InsertionPoint.AfterEnd,
      Retrieve("ClassScope(ProceduralWall)
            .ConsScope.LoadData"),
      Retrieve("CollapsePolylineSegments"),
      bindingExprList);
  }
}
```

Back to the example, from the XML in Listing 2 we got the aspect:

```
public class GeneratedAspect {
  public GeneratedAspect(){
   Inject(InsertionPoint.AfterStart,
     Retrieve("ClassScope(ProceduralWall)"),
     Retrieve("ProfileExtrusion.Profile"), null);
   Inject(InsertionPoint.BeforeEnd,
     Retrieve(
       "ClassScope(ProcedurallWall).ConsScope"),
     Retrieve("ClassScope(DataLoader)
              .MethodScope(LoadProfile)"),
     bindingExprList_1);
   Inject(InsertionPoint.AfterStart,
     Retrieve("ClassScope(ProcedurallWall)
              .MethodScope(BuildMesh)"),
     Retrieve("ClassScope(ProfileExtrusion)
              .MethodScope(Extrude)"),
     bindingExprList_2);
  }
}
```

The *binding expression* used to bind the fragment with the context can involve the use of constants, partial evaluation of the fragment and the bindings can reduce the code of the fragment (i.e., conditions with constant values) leading to better performance and working sets.

3.3 Pointcuts and Code Simplification

The dot separated path used in the previous examples to retrieve the annotations is not enough flexible to express the pointcuts necessary to perform simplification. Pointcuts must be expressed as annotation sub-paths of the available hierarchy; this can be boring, error-prone and not possible when the hierarchy is unknown (e.g., if dynamically generated as in our case). Often also the annotation positions and order in the hierarchy are unknown so wildcards are necessary to express patterns like *every X occurring in Y neglecting where they occur in the graph of annotations*. To this respect, we adopt the "." symbol to represent nesting between annotation scopes and "|" for siblings. "->" and "∗" to represent an unknown path to an annotation scope and multiple annotation scopes identified by common sub-strings respectively. The "[«annotation »]∗" is used to represent a sequence of arbitrary length for a given annotation. Considering the annotation hierarchy in the previous example admissible patterns are:

```
ClassScope(∗).∗Scope.Load∗ ClassScope(ProcedureWall).->
        ClassScope(ProcedureWall).->.LoadData
      ClassScope(ProcedureWall).->.[LoadData]∗
```

Each of these matches the same path on our example. Note that the "->" operator follows the blueprint loose arrow semantics closely [2].

With a description authored by DCC tools we can weave the aspects into ProceduralWall allowing computation of the final high quality mesh. Because we still need coarser *LOD*s we can use simplified version of the procedure instead of reducing the final mesh. Reduction means that the full code for the object generation must be executed followed by the decimation process as described in Sect. 2. In our approach we would simplify BuildMesh() method's behaviour to obtain a more efficient representation using specific rules. In this case we can replace the *lamppost* by two cuboids resulting in a more desirable reduction than that achievable through automated processing. Similarly the *pole* can be replaced by a cuboid and the *profile extrusion* replaced by a rectangle. To facilitate this we apply rules to select a pattern of annotation and replace it with another. Selecting the

```
ClassScope(ProceduralWall).
MethodScope(BuildMesh).->.Extrude
```

fragment and replacing it with

```
ClassScope(Rectangle3D).ConsScope
```

[5] Class, Method and Constructor can instantiate «Elem».

[6] http://blogs.msdn.com/clrcodegeneration/

	Procedural	3D Mesh
Vertex	2	52
Normal	0	43
UV Coordinates	0	52
Tangents	0	52
Pole Radius	1	0
Data Size in byte	**29**	**2180**

Table 1: Data Comparison

fragment will delete the `Extrude()`'s closure and inject the code to build a plane on a segment of the wall (the binding must be provided to set the wall height and the parameters for the rectangle construction). In the same way rules for replacing pole and the lamppost generation will be used to simplify the `BuildMesh()` method. Rules are able to target sequences as well as unique elements, using this we can further reduce the wall by targeting

```
ClassScope(ProceduralWall).
MethodScope(BuildMesh).->.(Pole|[Extrusion]*|Pole)
```

and substituting it by just

```
ClassScope(Rectangle3D).ConsScope
```

The `Substitution` operator is realised through the `Inject()` and `Delete()` operators on patterns discovered by the `Search()` operator. After retrieval this new fragment is injected before the beginning (`BeforeBegin`) of the original fragment which is consequently removed. Deletion is the last step to ensure that only the really unused variables are eliminated along with the fragment. Simplification of the resultant generator occurs before content creation, in this way simplified code is immediately ready to be used and a simpler version of the renderable mesh can be used whilst waiting for the fine version.

4. DISCUSSION AND CONCLUSIONS

The proposed aspect-oriented approach to procedural content design separates the content description from its generation, deferring the code generation until the instantiation of the target media by the application. At this time optimised code is built for *LOD*s preserving the peculiarity of the content being generated. Our focus on the algorithmic regard of procedural content is able to exploit features present in source data that may be lost after generation of the final media. Using our technique the performance of the content generation can be boosted through inspection, taking advantage of every facility the host can provide such as injecting an hardware accelerated implementation over a software alternative. Approaches focused upon final content are not able to dynamically produce other versions of the target media whilst maintaining relevant features without inclusion of additional information and implementation detail. In contrast our approach requires only the source data to generate multiple *LOD*s using a single source whilst maintaining features as specified by the media author. Simplification of code instead of data achieves the goal of faster loading of *LOD*s at smaller sizes than the original data representation.

In our scenario there is no need for the data to be fully generated in order to build the *LOD*s, we generate coarse representations using the simplified code directly. The code simplification rules are specific to annotation patterns and thus we can adopt ad-hoc strategies instead of general heuristic based reductions. This substitution uses the same set of fragments as the high quality aspect generation, there is no need for external sets of annotations to perform the rewrite process as can be required with grammar based approaches. Run-time grammar changes can be hard to manage, especially in cases where there is need to transform productions into terminals or vice versa. Our approach entirely avoids these complications.

Table 1 compares the data size of a block of the wall made by one section and two poles. The procedural approach requires 2180 bytes of mesh data to be created only in the video memory with the overhead of 29 bytes in system memory compared to the non procedural approach which requires 2180 bytes to be allocated in both system and video memory. Using a hardware accelerated approach to geometry generation, as the one provided by the Geometry Shader program in DirectX10[7], allows us to take this further and stream the procedural data itself to the video card in preference of the full mesh thereby reducing memory costs further achieving a total streamed data size of 29 bytes. The non procedural mesh can of course be reduced by removing run-time calculable data such as normals, tangents and uv coordinates however this will incur in recalculation costs and will require a secondary data set for topological information concerning the geometry. The procedural approach by its generative nature implicitly describes the topology removing the need for computation of this extra data.

5. REFERENCES

[1] W. Cazzola, A. Cisternino, and D. Colombo. Freely Annotating C#. *Journal of Object Technology*, 4(10):31–48, Dec. 2005.

[2] W. Cazzola and S. Pini. On the Footprints of Join Points: The Blueprint Approach. *Journal of Object Technology*, 6(7):167–192, Aug. 2007.

[3] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, and M. Tarini. Preserving Attribute Values on Simplified Meshes by Resampling Detail Textures. *The Visual Computer*, 15(10):519–539, Dec. 1999.

[4] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli. C-BDAM – Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering. *Computer Graphics Forum*, 25(3):333–342, Sept. 2006.

[5] H. Hoppe. Progressive Meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'96)*, pages 99–108, New Orleans, Louisiana, USA, Aug. 1996.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, LNCS 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.

[7] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool. Procedural Modeling of Buildings. In *SIGGRAPH '06: Proceedings of the 33rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 614–623, Boston, MA, USA, Aug. 2006. ACM.

[8] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of Triangle Meshes. In *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pages 65–70, New York, NY, USA, July 1992. ACM.

[9] A. Vasseur. Dynamic AOP and Runtime Weaving for Java- How Does AspectWerkz Address It? In R. E. Filman, M. Haupt, K. Mehner, and M. Mezini, editors, *Proceedings of the 2004 Dynamic Aspect Workshop (DAW'04)*, pages 135–145, Lancaster, England, Mar. 2004.

[7] Shader Model 4 at `http://msdn.microsoft.com/en-us/library/bb509657(VS.85).aspx`.