

ZLang: A Scripting Language for Digital Content Creation Applications

MohamedYousef

Ahmed Hashem

Hassan Saad

Khaled Hussain

Faculty of Computers and Information, Assiut University, Assiut, Egypt

ABSTRACT

Digital Content Creation (DCC) Applications (e.g. Blender, Autodesk 3ds Max) have long been used for the creation and editing of digital content (e.g. Images, videos). Due to current advancement in the field, the need for controlled automated work forced these applications to add support for scripting languages that gave power to artists without diving into many details. With time these languages developed into more mature languages and were used for more complex tasks (driving physics simulations, controlling particle systems, or even game engines). For long, these languages have been interpreted, embedded within the applications, lagging the UIs or incomparable with real programming languages (regarding Completeness, Expressiveness, Extensibility and Abstractions). In this paper, we present a high level scripting language (ZLang) and a DCC Engine that addresses those problems. The language can be interpreted, compiled, extended in C/C++ and has a number of constructs, and optimizations dedicated to DCC domain. The engine provides geometric primitives, mesh modifiers, key-framed animation and Physics Simulations (Rigid Body, and Cloth Simulations). The engine is designed and implemented as a library so it can be used alone or embedded.

General Terms

Digital Content Creation, Computational geometry, Scripting Languages.

Keywords

Three-DimensionalGraphics and Realism, modeling Packages, Methodology and Techniques-Languages.

1. INTRODUCTION

Digital Content Creation (DCC) is a general term usually used to refer to the creation and editing of any form of digital content (e.g. images and videos). Since the dawn of computer graphics, DCC software has played an integral role in automating the process and simplifying it to artists with no scientific background. The need to manage many details in a repetitive and accurate way, lead to the design of Domain Specific Languages (DSLs) targeting the DCC domain, which provided an intuitive interface User Interface (UI) at time, with special interest given to being accessible to artists with poor/no scientific background in computer graphics. Gradually these languages developed into more complete and mature languages and were utilized in more complex DCC tasks (driving physics simulations [1], controlling particle systems [2], simple/experimental Mesh processing algorithms or even game engines).

A Challenge faced by all DCC languages is how to be simple and easy to use for non-experienced programmers without compromising being general and familiar to regular, experienced programmers. A practice used widely by today scripting languages [3] in general, is to be dynamically typed, provide very high level constructs, extensive standard library,

and a powerful extensibility mechanism. A similar means have been adopted in DCC languages with problems arising particularly in being general and extensible, thus supporting wide spectrum of users. This is specifically true for commercial, custom-users driven packages.

Two approaches were used to implement those languages. Either build a DSL that wraps the UI (like MAXScript[4] and Maya Embedded Language (MEL) [5]), or use an existing popular language and write a set of extensions to it to wrap the UI and embed it (like Blender [6]). In practice, both those solutions suffer. The first method produces languages lacking being general, competitive languages and are generally very inefficient. The second method has problems arising from not being dedicated in first place for that kind of applications. So, they lack expressiveness facilities (like dedicated constructs) that facilitate supporting the DCC domain, also it is very hard to optimize these languages for specific situations frequently encountered in a domain like DCC.

In this paper, we present a system that addresses those problems. A high level scripting language (ZLang) and a DCC Engine, the language can be interpreted, compiled, extended in C/C++ and has a number of constructs and optimizations dedicated to DCC domain. The engine provides geometric primitives, mesh modifiers, key-framed animation and Physics Simulations.

ZLang has three major design decisions that make it distinct from existing systems, with the goal of solving practical problems that existing systems suffer from. First, ZLang is a general purpose programming language oriented to DCC applications. Thus, it has all tools assisting it as a general purpose language while complementing this with DCC specific constructs and optimizations (which can't be supported by embedding Python in Blender, as Python is a general purpose language that can't provide any intrinsic support for a particular domain), and supporting library which interfaces with the DCC engine. This makes ZLang usable for DCC complementary deployments not originally planned for at time of designing. Second, ZLang is imperative language, most of the previous research systems are based on functional languages ([7], [8], [9]). A strong feature of imperative languages is that they are familiar to every programmer, anybody with previous experience with any of main-stream production languages (e.g. C++, Java, C#, Python) will find that ZLang is very intuitive and easy to learn and use. This coupled with special DCC constructs, makes ZLang Syntax perfect for its particular domain and user-base. Third, ZLang is cross-platform, free, and open source. Unlike current famous systems like MAXScript and MEL that are closed source and commercial. Also ZLang scripts are dependent only on the ZLang interpreter, which itself can be both embedded in other applications and extended by writing modules to it.

In following sections, we first provide an overview of related work in computer graphics literature. Then, we describe

ZLang syntax and semantics, its type system, programming paradigm, and methods of extensibility. After that, we describe the ZLang DCC Engine, how we construct primitives, apply modifiers, texture map objects, and produce key-frame and game physics animation. Finally, we describe our C++ implementation of ZLang, its system architecture, and our future plans for it.

2. RELATED WORK

The literature describes many previous DCC systems which used a programming language (a full language or just a limited sub-set) as its user interface. Here, we try to shed light on some of the most related and influencing to our work in ZLang. Before we dig in literature, we mention current production-ready systems that had direct influence on ZLang. MEL is a scripting language that is syntactically similar to TCL and Perl, used for automating tasks in Maya [10]. MEL is not object oriented, and lacks advanced features such as associative arrays. Very few improvements have been made to it in recent years. Autodesk 3ds Max [11] uses MAXScript for the same role of MEL. In contrast to MEL, MAXScript is object oriented and has many high level data structures suited to DCC domain. The main drawbacks of MAXScript are: it's closed, commercial, tied to 3ds Max, and lacks the expressiveness and generality of main stream scripting languages such as Python [12] and Ruby. Python is a main stream general purpose scripting language, used extensively in many fields and platforms. Due to its stability and proved success, it was incorporated in Blender [13] as a scripting language. Its main drawbacks, are the drawbacks of deploying a general purpose language for a specific domain: no compiler support (e.g. no domain specific optimizations), and no dedicated constructs.

2.1 Procedural Modeling

Procedural modeling is an umbrella term for a number of techniques in computer graphics to create 3D models and textures from sets of rules. It ranges in complexity from parameterizing simple algorithms to full programming languages, and remains as root of utilizing power of automations in creating digital content such as describing complex shapes or animations that are too tedious to describe explicitly (through other interfaces such as a GUI).

L-Systems are a common solution to modeling plants [14], [15], and can also be used for streets and buildings [16]. These systems are based on rules for replacing string parts to derive a high-level description of the model, for example the skeleton of a plant, and generate the geometry in a second step. An alternative technique for modeling buildings is shape grammars [17], [18], [19]. The drawback of grammars is that they are usually limited to a specific class of models and they require a lot of training and time for experimenting. Furthermore, it is not obvious how to parallelize the rewriting of context sensitive grammars to make use of modern multicore CPUs. John Snyder introduced the GenMod system for generative modeling [20]. The system produces 3D shapes from several curves without an intermediate step. It is based on a C interpreter with overloaded operators. Thus, it has variables, arrays, loops and formulas in infix notation. More recently, Sven Havemann implemented a similar system that employs a stack-based postfix notation to avoid the need for variables and a parser [21]. His system is called GML, for Generative Modeling Language. GML's applications focus on architecture and its most interesting feature are programmable gizmos. Gizmos are special points that allow the user to edit the parameters of a primitive by moving a handle in the

viewport. GML's gizmos allow the user to define how the gizmos are mapped to parameters of an object.

Several authors [22], [23], [8], [24] have taken procedural modeling to another level by creating procedural modeling (and shading) languages. The procedural modeling languages are generally based on existing languages. This allows the procedural modeling language to inherit the functionality and constructs of the underlying language. Each of the procedural modeling languages presents the programmer with different tools and data structures. The Renderman Shading Language [25] is a procedural shading language that embodies several of the desirable aspects of a procedural modeling language. Some procedural modeling facilities, such as displacement shaders, are available. The shading language provides domain specific functions and operators which facilitate commonly used shading operations. The operators provided by procedural modeling languages include union, intersection and subtraction [22], [23], the synthesis of shapes on existing geometry [26], [27], Weathering of geometry [28], [23], cutting operations [29] and extrusion [8], [16], [24], [30]. Model representations include both surface and volumetric models. Surface models can be explicit sets of polygons, implicit surfaces [22], patches, quadric surfaces and spheres [8], [24]. Volumetric representations include tetrahedral meshes and signed distance fields [23] and particles [2], [29].

2.2 Animation

The literature describes many previous animation systems, often with a procedural component. Early systems, such as Scripts and Actors [7], provided a language for defining modeling and animation. Independent actors control different visible elements in a scene and are invoked every time-step. MENV [24] controls animation by changing the value of *avars*, articulated variables which look like regular scalar variables and as such can be used as arguments to functions, as conditions in if/then statements, etc. However, unlike typical variables, the value is externally defined (from the program), and time-dependent. The value of an *avar* at the current time is determined by interpolating a spline through a set of interactively, or procedurally, specified key frames. Improv [31] combines two procedural components: a Behavior Engine for deciding what actions a character should perform, and an Animation Engine for controlling these movements. Actions can be layered and movement transitions are normally specified using either sinusoids or different frequency noise functions [32]. AL [8] is another language that builds on concepts from MENV and others and extends by adding a generalization of *avar* concept known as articulated functions (*Afuncs*). ZLang on the other side implements another means to achieve full control on key-frame specification, this is done through the animate construct which creates a context in which all actions (variable setting, function call, etc.) have effect on particular specified key-frame. AL is based on Scheme. Another similar language, Fran [33] is based on Haskell.

3. ZLANG

ZLang is a dynamic-strongly typed, hybrid paradigm (supporting both object oriented and procedural paradigms), memory managed language. These combinations of features were carefully picked to support goals of the language related to expressiveness, completeness and simplicity.

ZLang is divided into two main parts: the Interpreter and the DCC Engine. Here, we describe the details of the first of these, focusing on the syntax, type system, extensibility

model, and design decisions of the language. An example for general ZLang code is given in Listing 1.

```
fn ToBase number , base =
(
  alpha =["0","1","2","3","4","5","6","7","8",
    "9","A","B","C","D","E","F","G","H","I",
    "J","K","L","M","N","O","P","Q","R","S",
    "T","U","V","W","X","Y","Z"]

  str=ZString()
  while number>=1 do
  (
    str.Insert( alpha[number%base] ,0)
    number=number/base
  )
  str
)
print("Enter Base : ");
b = if (( b=ZConvert.AsInt(Read()) )<=36 &&
  b>=2 )
  then b
  else(printL("Base corrected to 2");2)
print("Enter Number : ");
n = ZConvert.AsInt(Read())
printL(ToBase(n,b))
```

Listing 1: ZLang code that converts a number from decimal, to any other base from 2 to 36.

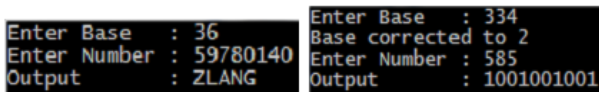


Figure 1: Output from code Listing 1.

3.1 ZLang Syntax and Semantics

The syntax of ZLang resembles in a number of aspects scripting languages like Python and alike DSLs like MAXScript. This stems from the need of ZLang to be both familiar to users and programmers familiarized with current technology.

ZLang is an Expression-based Language; that is every statement in ZLang is an expression and every expression must have a value which can be simply, the value associated with name for a variable, or return from a function or even the last statement in a group of expressions. Groups of expressions are enclosed between two parentheses "(" and ")". This is one of the greatest sources of flexibility of ZLang.

As for control flow constructs, ZLang provides: If and Case as decision taking constructs. Case comes in two flavours: One that is static, the usual form provided by languages, and hence allowing for optimizing it at run-time. The other flavour is a dynamic one, accepting any type and allowing expressions as case values. For, While, Do...While and exit/continue are provided as looping constructs. For comes in a variety of ways, either the usual direct form with start, end, and increment values, or looping over elements of a List (more on ZLang built-in types in subsequent sections), also a collect option groups results from each loop in a List. exit/continue statements can be applied over a chosen number of enclosing loops.

Functions in ZLang have a dynamic number of arguments. Arguments can have default values, and return values can be specified explicitly through a return statement or implicitly as the value of last expression. An important feature of ZLang functions is that they are first class data-types. So, they can be

passed around (e.g. for using as callbacks) and can be constructed in-place, without names (however they aren't complete lambdas, as they don't preserve state).

As ZLang is an object-oriented language, a programmer can construct classes containing functions, and variables and later instantiate objects from these classes. Single inheritance is supported. After instantiation every object has an independent symbol table accessed through the dot "." operator. Thus any properties of a specific object can be modified, without affecting other objects belonging to same class. ZLang comes with an object oriented standard library, covering a wide range of topics, written in C++ as an extension module to the interpreter. The ZLang DCC Engine is exposed to the language through a rich OO API.

ZLang is statically scoped, every group of expressions represent a specific scope, except for the body of the If and Case to give programmer flexibility to define new variables in them. Every scope is linked to the enclosing scope. Every object has an associated scope. A hash table is used to associate names with values in a scope.

ZLang is memory managed, Mark and Sweep [34] is used to track unreferenced regions of memory (e.g. objects that goes out of scope). As opposed to Reference Counting, Mark and Sweep is unaffected by cyclic references, which is an important feature for an object oriented language and the fact that it's a non-deterministic algorithm doesn't affect ZLang operation.

ZLang supports single and multi-line comments. Line terminals (e.g. semicolons) aren't needed to separate lines. White spaces works as a separators, they can even be neglected when token end can be determined unambiguously from its lexical structure (e.g. a number).

3.2 ZLang Type System

A central component to any language is its type system. This single feature plays integral role in the usability, expressiveness, interoperability and extendibility. ZLang is a dynamic-strongly typed language [3]. We choose a dynamic type system [35] because of its expressiveness and for relieving user from burden of static typing (at cost of optimizability). We choose a strong type system to enable a subset of type-based optimizations, and keep language behavior much predictable.

Since ZLang is strongly typed, every reference to a variable is mapped to one of the set of ZLang's internal data-types. Internal data-types are most basic data-types of ZLang over which other types (through OO) and operations are built. ZLang adopts the idea of providing some high-level internal data-types (implemented in other scripting languages like Python and Ruby) for the convenience of the programmer. ZLang's internal data-types are *Integer*, *Float*, *Boolean*, *String*, *Function*, *List*, *Dictionary*, *Matrix*, *Instance*. We call the type exposed to language as union of these a ZTvar and a pointer to it a ZTvarp.

Function represents the function basic data-type giving functionality of a first class function data-type previously described. List is a list of elements. Since, ZLang is dynamically typed; the List elements can be a mix of a variety of types. Thus a multi-dimensional list is a list of lists. Dictionary is a hash table indexed by strings, each pointing to a ZTvar. Matrix is used for representing matrices and vectors and their operations, this type came from the necessity in DCC to be not just a module, but built-into the language with syntactic support. Instance is an instance of any class in

ZLang. Operations on simple data-types (Integer, Float and Boolean) are carried by value, while on the rest of types they are carried by reference. Respective meaningful operators are overloaded for each type.

3.3 ZLang Extensibility

Another central feature for today's scripting languages is the ability to make use of existing components written in other languages (most importantly C and C++). This is usually implemented as an extension module to the language that works as glue code connecting the external library and the language. The amount and complexity of glue code and type mapping to and from language's type system, largely determines how extensible the language is.

Another factor also is how much of the language is constructable and modifiable from within an extension module. Generally, ZLang can be extended in two ways: by adding a function, or a class.

Modules can be either: statically linked with the interpreter, so that they are permanent part of the interpreter, or dynamically linked at runtime (the module is loaded at runtime upon request).

Functions

The first way to extend ZLang is to add a function to ZLang that is accessible from global scope. This can be done from C++. For functions to be invocable from ZLang they must adhere to a specific signature which takes an input as a vector of ZTvarp and returns a ZTvarp. This way, we can simply mimic void functions (null ZTvarp), multi-return (a ZTvarp of type List), variable number of arguments (as vector size is dynamic and query-able) and default values for arguments (through querying types and number of arguments). Listing 2 shows a template for such a function in C++.

```
ZTvarp ZMyFunction(ZTvarS var)
{
    //do some stuff with var if needed
    //allocate and return a ZTvarp if needed
}
```

Listing2: Template for a C++ function that is invocable from ZLang.

Adding Classes

The second way to extend ZLang is to add a complete abstract data-type to the language, with all its set of methods and data members. This is specially crafted to be done seamlessly easy from C++. To write a C++ class that is exposed to ZLang, all methods have to be defined according to rules previously stated in function section, and all data members to be exposed, must be ZTvar's. After that, an initialization function is used, to set names of all methods, and variables to be used from ZLang as class members. This pipeline simplifies and greatly reduces effort to export complex classes and data structures so as to be used from ZLang. Listing 3 shows a template for such a class in C++.

```
class ZMyClass :
    public ZTBaseObject<ZMyClass>
{
public:
    ZTFloat x,y,z;
    static void Init()
    {
        /*
         any application-specific initializations
         add functions and variables to symbol
         table.
        */
    }
    ZMyClass (ZTvarS inp)
    {
        /*
         handle class constructors when
         called from ZLang.
        */
    }
    ...
    ZTvarp ZMyFunction(ZTvarS var)
    {
        /*
         do some stuff with var if needed.
         allocate and return a ZTvarp if needed
        */
    }
    ...
}
```

Listing 3: Template for a C++ class that can be used from ZLang.

Standard Library

Existence of a well-designed Standard library is a central feature to the success of any scripting language. Pre-packaged standard libraries (an idea that first demonstrated its success with the appearance of Java [36]) greatly reduces effort on part of programmer and increases efficiency of the development life cycle.

Currently ZLang Standard Library (ZSL) includes I/O, Mathematics, File-System, and Matrix libraries. The DCC Engine itself is part of the ZSL, a set of wrapping classes is written for every class in the engine, and designed to have as simple and intuitive API as possible.

The output from ZLang can be: a model, a key-framed animation, or a physics simulation. There are two controlling objects KeyFrameAnimation and PhysicsSimulation these are used for managing both types of animation. Every primitive is an independent class that can be instantiated and added to the controllers (with a variety of convenience constructors). Every Modifier is an independent class. Modifiers can be either applied directly on primitives changing the original mesh, or calculated progressively as in key-frame animation using the animate construct. Texture and light effects can be applied when adding any primitive to the scene through any controller.

4. ZLang DCC Engine

The second integral part of ZLang is its DCC Engine. The engine provides geometric primitives, mesh modifiers, key-framed animation, physics simulations and visualization effects (Texture and lighting). The engine is implemented in a modular fashion. Every one of previous features is an

independent module that interacts with others through a predefined reusable interface.

In each of the following sections, we present a primitive class supported by the engine along with an example demonstrating how an instance of that class is implemented (e.g. how a Box which is a geometric primitive is constructed). A list of all primitive classes and their instances is presented in Figure 2.

4.1 Geometric Primitives

We use half-edge data structure [37], [38] to store meshes in ZLang. Half-edge data structure's biggest feature for our usage is the efficiency of local updates and queries. This enables for a very fast construction of primitives, application of modifiers, and also practical export and import time of meshes.

To construct primitives, we employed two techniques. The first one was to use Euler operations directly on the half-edges and progressively sculpt the mesh. The second method is the standard way of defining the mesh step by step by specifying the faces of the mesh vertex by vertex (we called it delegate method). While the second way is generally high-level and simpler, for some shapes it's simpler and more efficient to construct primitives directly using Euler operations, because of the complexity associated with correctly connecting shape points into faces. The engine supports following primitives: Box, Cylinder, Cone, Pyramid, and Plane (through Euler operations), and Sphere, Spindle, Torus, Spring, Tube, and Lathe (through vertex-face setting). Associated with each primitive is a Modifier stack, at each frame all modifiers in stack are all applied to the primitive in order.

As an example of Euler-operations based primitives we use the Box, Figure 3 illustrates the series of Euler-operations involved in constructing a Box.

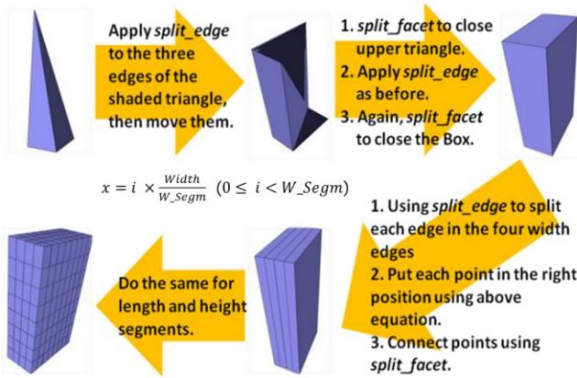


Figure 3: Steps for sculpturing a box through Euler-operations.

As an example for delegate-based primitives, we demonstrate constructing a sphere. First we generate sphere points ;since a sphere can be described as a group of circles with increasing and then decreasing radii on the direction of increase of Z axis, the x , y , z values for sphere points can be generated by following equations; Where r is the sphere radius. φ is constant and θ changes from 0 to 2π for every z value

$$\begin{aligned} rad &= r \sin \varphi \\ x &= x_0 + rad \cos \theta \\ y &= y_0 + rad \sin \theta \\ z &= z_0 + r \cos \varphi \end{aligned}$$

After that we connect these points into faces, we will use quad faces only for the case of the sphere, we use algorithm in Listing 4. A progressive construction of a sphere is shown in Figure 4.

```
Set n = num_side_segments
for e=0 to n/2-2
for i=0 to n
/*here we construct face by providing
indices of it's vertices*/
Sphere.AddFace ( e*n + (i+1) % n + n, e*n+
i + n, e*n+ i, e*n + (i+1) % n )
```

Listing 4: Constructing sphere's faces.

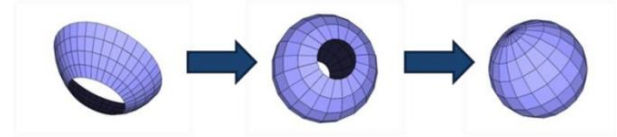


Figure 4: Step-by-Step sphere construction in increasing values of Z and φ , a final step not discussed here for brevity is the adding of final points that connect upper and lower holes that will be left in sphere (representing zero-radius circles).

4.2 Mesh Modifiers

Modifiers are indispensable tools in the hands of the designer in any DCC applications. Modifiers implementation in ZLang relies on Euler operations to perform face level modifications and queries. Modifiers can be applied on primitives on two ways: the first way is to change the underlying mesh permanently with no way back, this optimizes on space and time but loses flexibility, and the other way is to add the modifier to the modifier stack of respective primitive. A single modifier can be applied to many primitives concurrently, resulting in similar modifications on all.

Two types of modifiers are implemented in ZLang. The first is global modifiers [39], these modifiers work on all vertices of the mesh, modifying them one by one (the input to each iteration of execution is a vertex in the mesh) finishing with the whole shape or part of it transformed in a certain way (actually this is a space transformation, we end up with the image of our original mesh in the transformed space as seen from our space). Our global modifiers are Bend, Bulge, Twist, Taper, Skew, Spherify, Cylindrical Wave, Linear Wave, Squeeze, Stretch, and Noise. The second type is Facet modifiers, these either modify a certain facet, or work on a facet by facet basis on the input mesh (the input to each iteration; is a whole facet not a single vertex). Our facet modifiers are Outline, Extrude, Bevel, Triangulate and Smooth.

As an example for the global modifiers we discuss the Taper modifier, which has an effect like bumping the shape. It takes as input the amount of tapering, the tapering axis, and tapering limits which define part of mesh where tapering takes effect. Since this is a global modifier, each vertex is treated individually, and we have an equation for calculating the position of vertex after the application of the modifier, based solely on its current value. For taper we use following equations

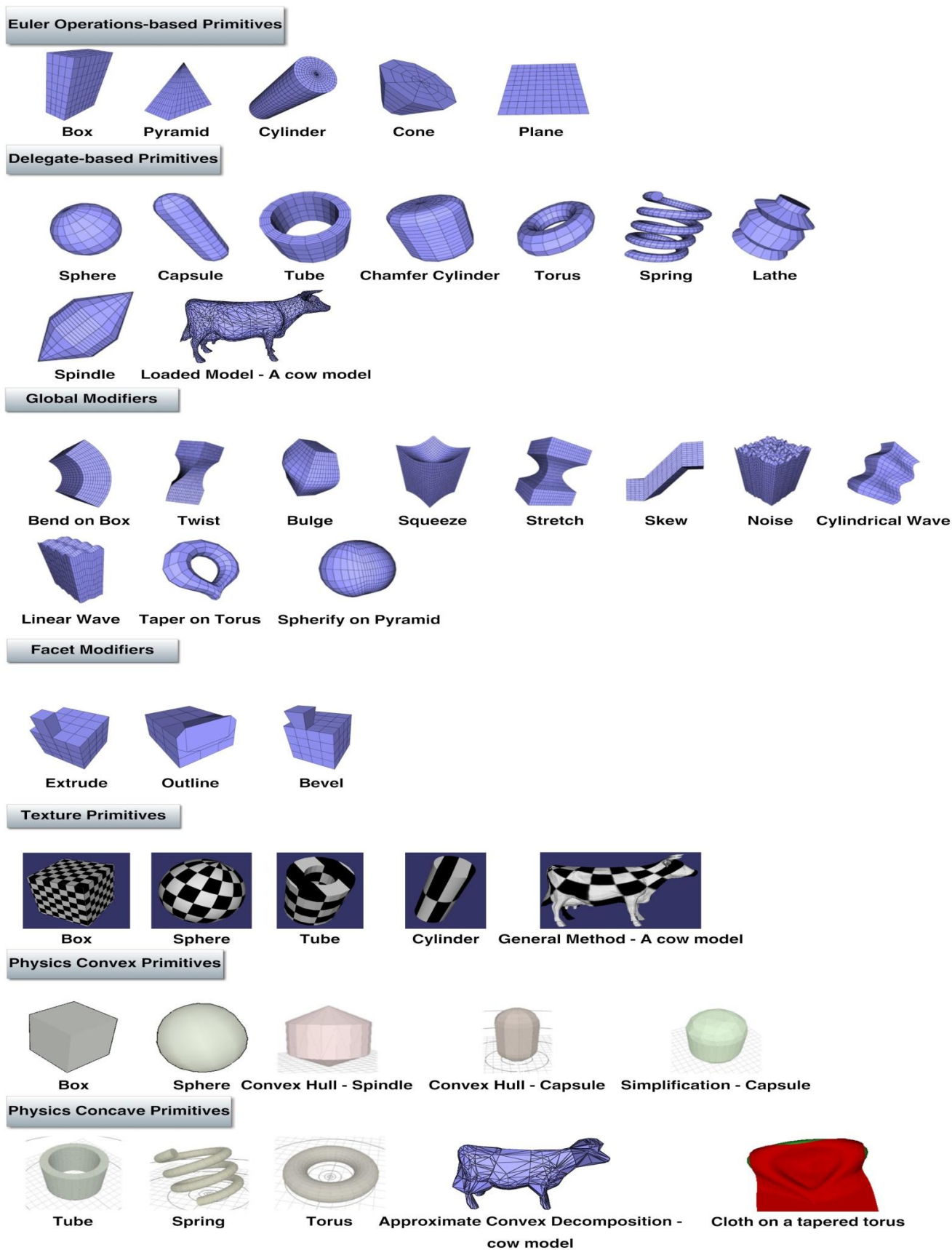


Figure 2:DCC Engine Primitives. The figure illustrates most of the primitives supported by ZLang DCC Engine, categorized by their types.

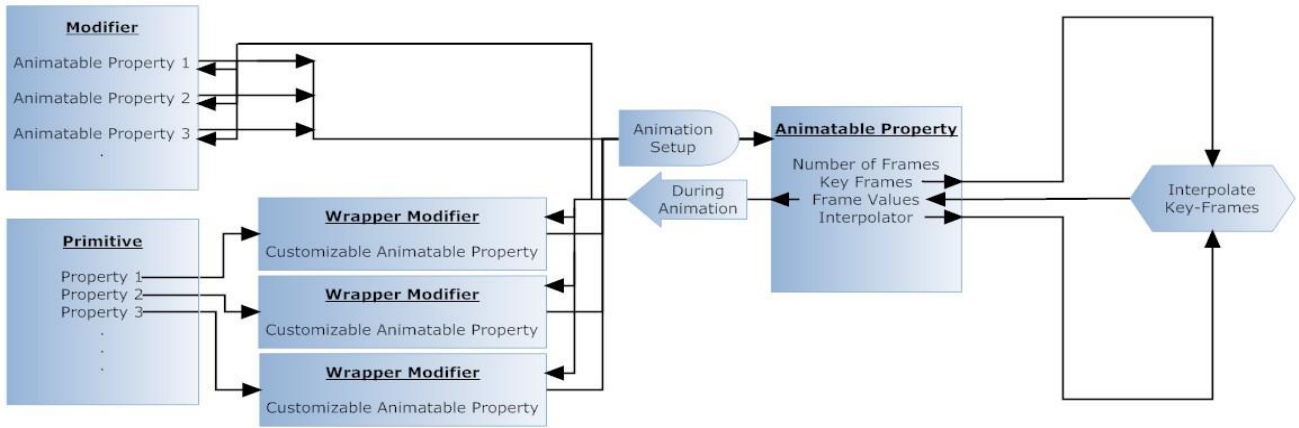


Figure 6: The Key-Framed animation pipeline in ZLang, it consists of two stages, the first stage (left to right) fills the AnimatableProperties with appropriate key-frame values, and in the second stage (right to left) these values are interpolated to produce value for every frame, these values are set during animation at every frame and their effect applied.

$$S = \begin{cases} TM \times UL / (wMax - wMin) & \text{if } w' > UL, \\ TM \times LL / (wMax - wMin) & \text{if } w' < LL, \\ TM \times w' / (wMax - wMin) & \text{if } LL \leq w' \leq UL \end{cases}$$

$$\begin{aligned} u &= u' + v' \times S \\ v &= v' + v' \times S \\ w &= w' \end{aligned}$$

Where **W** is the Taper Axis, **U** and **V** are the other two axes and $U \neq V \neq W$. u' , v' and w' are the old **U**, **V** and **W** position of each vertex in the mesh and u , v and w are the new values. TM is the taper amount. UL and LL are the upper and lower limits of applying the taper modifier. $wMin$ and $wMax$ are the minimum and maximum value of the **W** component of any vertex in input mesh. Figure 5 illustrates the result of applying a taper modifier with amount 1.5 on **Z** Axis on a sphere.

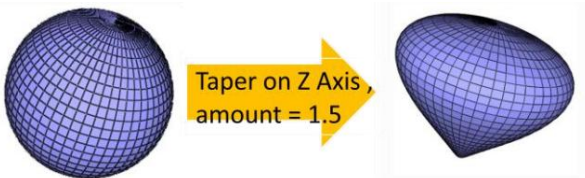


Figure 5: The effect of applying the taper modifier on a sphere.

4.3 Texture Mapping

Texture mapping [40] is the process of adding details, textures to the surface of a 3D object. The core of the process is a projection operation from 3D space (the 3D object) to 2D (the texture). So, every (x, y, z) point is given a (u, v) texture. This process improves a lot on realism of output object, and is used also for adding detail and improving quality of a low polygon meshes.

ZLang provides two ways to texture map an input 3D object. The first is a general algorithm [41] that works with arbitrary shapes and achieves plausible mapping results. The need for a second method, stemmed from the fact that the first method doesn't produce good results with all our primitives and nongeneral method will. So, we used specialized algorithms for some of our primitives to achieve optimal results for them. We provide specialized texturing ways (calculating uv values

for each vertex) for following *Box*, *Pyramid*, *Plane*, *Cylinder*, *Sphere*, and *Tube*.

As an example for specialized texturing algorithms, we demonstrate texturing a sphere. The most important part of texturing process is the algorithm that transforms (x, y, z) coordinate into (u, v) coordinates. For the sphere case we use following equations

$$\begin{aligned} v &= \arccos\left(\frac{p.z}{radius}\right) / \pi \\ u &= \arccos\left(\frac{p.y}{radius \times \sin(v \times \pi)}\right) / (2\pi) \end{aligned}$$

Where p is any vertex on sphere and $p.y$ is the **Y** component of vertex p . Figure 7 shows result of texturing a sphere.

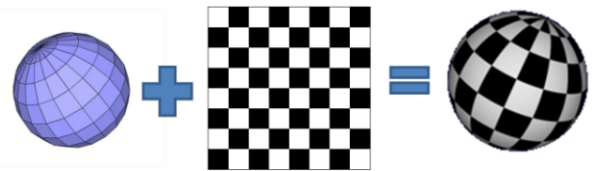


Figure 7: Texturing a sphere using a checker-board texture.

4.4 Animation

Producing animation from the shapes that we generated in previous sections is a key DCC application and is a source of much of appeal to the DCC industry. ZLang provides two methods to animate objects. The first is key-framed animation, where animation illusion is achieved through interpolating user-supplied values, to get intermediate frame-wise values for specific properties. The change of these values creates animation. The second way approximates laws of Physics and applies them to simulation objects to create realistic looking animation.

4.4.1 Key-Framed Animation

To create key-frame animation, the user supplies values for specific variables that control shape, or properties of objects at specific key-frames. The system then interpolates to get values at rest of frame and then creates the animation by setting each variable's value at the specific frame. In ZLang a user supplies values for what is called AnimatableProperties inside the animate construct. AnimatableProperties span three

regions in ZLang: they are either a modifier property (e.g. a Box), or scene node property (rotation of a node in the scene). After setting those values by user, the system interpolates to calculate the value of every AnimatableProperty at each frame (within range of animation), then the system is responsible for setting values for AnimatableProperties at animation run-time and forcing its application (e.g. re-apply modifier for an AnimatableProperty associated with modifier, or re-construct primitive for an AnimatableProperty associated with primitive properties). The whole pipeline is illustrated in Figure 6.

4.4.2 Physics Animation

To create Physics animation the pipeline goes in following steps. First, we set up the properties of world (e.g. gravity, friction). Then, we calculate and determine properties of shapes to be added to simulation, most importantly a bounding volume representation, and supply these to a physics engine that will be used for applying physics laws, and calculating object's translation and rotation at each frame (also position of each vertex for soft-body and cloth simulation). Lastly, in simulation loop at each frame the engine is queried to get data about objects and interaction from users are supplied to it (e.g. objects added, forces applied).

The challenge thus, is to correctly define arbitrary meshes to the physics engine. The idea is that these primitives should provide various options that, either balance speed and accuracy, or go only for one of them. Three groups of types of physics primitives are provided by the ZLang DCC Engine. These are convex, concave, and soft primitives.

Convex primitives are the most efficient (due to existence of optimized algorithms for collision detection of convex shapes), and can be used even for concave meshes

```

Set cxS as convexShape
Set vertCount = 12
for i = 0 to Num.Side_Segs * Num.Height_Segs
    Set face = First_face ( Input.Mesh )
    for j = 0 to i
        face++

    Set verts = Array [vertCount].
    Set h = 0
    for j = 0 to Num.Height_Segs
        Set n = first_half_edge ( face )
        for k = 0 to 6
            verts[h] = Point( n.vertex.x, n.vertex.y, n.vertex.z )
            n++
            h++

    Set u = i / Num.Side_Segs
    Set end = 2* Num.Side_Segs * Num.Height_Segs
              + Num.Side_Segs * Num.Cap_Segs
              -(2 * u + 1)* Num.Side_Segs

    for l = 0 to end
        face++

cxS.ConvexStack.Push(verts)

```

Listing5: Pseudo-code for tube convex-decomposition.

angle of Bend modifier) or a Primitive property (e.g. width of to approximate them to the physics engine. There are four kinds of primitives which can be represented in physics world as convex shapes these are *Box*, *Sphere*, *Plane*, *GeneralConvex*. The first two depend on bounding volumes [42], for the plane we use PCA [43] to fix a plane to the mesh. The last one is a general convex primitive used to approximate arbitrary shaped convex or concave meshes. Two methods are used for these, the first is by calculating an approximate (fast to calculate) convex hull of input mesh using quick hull algorithm [44], and the second method is using a geometry simplification algorithm [45], [46] that tries to reduce the number of edges in the input mesh without changing its shape.

Concave primitives, are the other less efficient option that preserves shape properties of concave primitives, like holes, as efficiently as possible. Two methods are provided for these primitives. The first method is a general algorithm for arbitrary concave meshes, the algorithm is Approximate Convex Decomposition [47], which tries to split a given concave mesh into a number of convex shapes and feeds them to the physics engine, as a connected group of convex shapes. The second method, utilizes our information about the structure of our primitives to construct a variable quality, near optimal convex decomposition of our concave primitives, the Tube, Spring, and Torus.

Soft primitives are those that represent meshes given special meaning and special prosperities in physics world. They include, Cloth, and Soft bodies, they can represent both convex and concave meshes.

As an example for convex-decomposition of ZLang concave primitives, we demonstrate decomposing a Tube. We use the algorithm in Listing 5, an example result is presented in Figure 8.

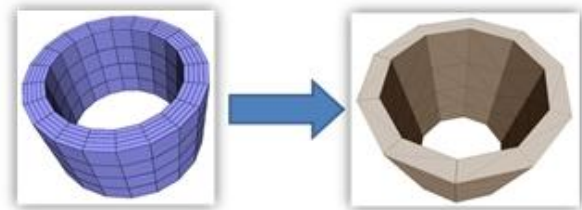


Figure 8: Result of convex decomposition of a tube, the right picture represents the tube as perceived by physics engine.

4.5 Scene Management

In order to manage and design a multi-object scene, special interest must be given to how to store pose information for each object, and how to efficiently draw the scene afterwards. We employ a scene graph data-structure for storing scene data. Every mesh is stored as a special scene node called PolyhedronNode, whose pose is controlled by a transformation matrix. The scene management component is also responsible for initiating scene update every frame (by calling update procedure of every object depending on type of animation applied on it), controlling frame rate, and responding to user input. Another important aspect of scene management is optimizing the drawing process, so that any object that is not currently in the viewing frustum is culled from scene.

PolyhedronNode stores object pose (with its Animate-able Properties), color, texture, and half-edge data-structure. On

drawing, every face of the half-edge is drawn as a separate

polygon. To draw a non-triangular wire-frame we iterate over

```

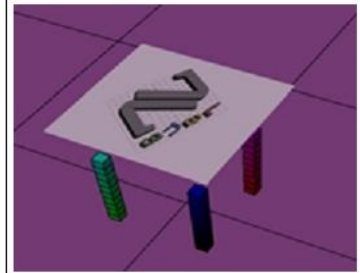
fn fallingClothOverArmature x,y,z =(
    newScene = PhysicsSimulation() ;
    planePrimitive = Plane(300,[0,0,0]) ;
    planePrimitive.Color(ZPoint(0.6,0.3,0.6))
    planePrimitive.setPhysActor(newScene,newScene.ZPhysPLANE)
    ;
    newScene.AddRigid(planePrimitive) ;

    clothPlane = Plane(50,100,[0,0,5*z])
    clothPlane.setWire(false)
    clothPlane.setPhysActor(newScene,newScene.ZPhysCLOTH)
    clothPlane.setPhysThickness(3.0)
    newScene.AddCloth(clothPlane,"Inversed Logo.jpg") ;

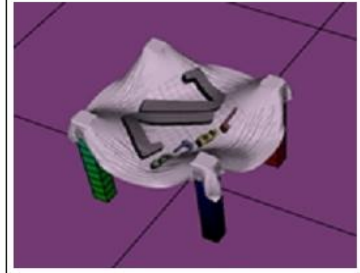
    for k = 0 to x by 8 do
        for i = 0 to z do
            for j = 0 to y by 8 do(
                box = Box(4,[k*4-((x-1)/2.0)*4, j*4-((y-1)/2.0)*4,i*4+2]) ;
                box.Color(ZPoint(k*0.9/10,j/10,i*0.8/10))
                box.setPhysActor(newScene,newScene.ZPhysBOX)
                box.setWire(true) ;
                newScene.AddRigid(box) ;
            )
        newScene.View() ;
    )
fallingClothOverArmature(9,9,9)

```

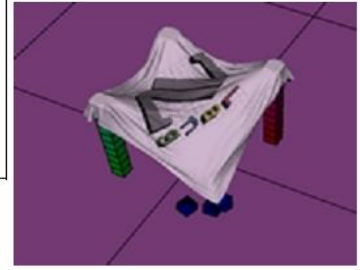
Listing 6: ZLang code demonstrating cloth and rigid-body interactions in a physics simulation.



Frame 1



Frame 100



Frame 150, after throw-

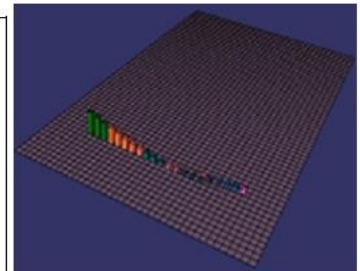
```

newScene=KeyFrameAnimation() ;
planePrimitive=Plane(600,900,4,6,[190,310,0]) ;
planePrimitive.setWire(true)
planePrimitive.Color(ZPoint(0.9,0.7,0.9))
newScene.AddPrimitive(planePrimitive) ;

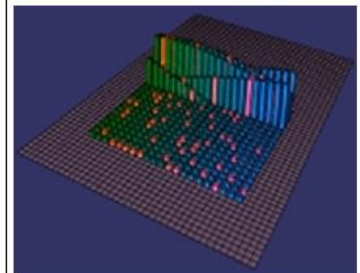
for j=0 to 30 do
    for i=0 to 20 do
        (
            box=Box(10,[20*i,20*j,5])
            box.Color(ZPoint(j*0.9/10,7/10,i*0.8/10))
            box.setWire(false) ;
            newScene.AddPrimitive(box) ;
            v = if j%2 ==0 then i
                else (20-i)
            animate on
            (
                at time ( 1+v+j*10)
                (box.Height(10) ; box.Position([20*i,20*j,5]) )
                at time (20+v+j*10)
                (box.Height(100) ; box.Position([20*i,20*j,50]) )
                at time (50+v+j*10)
                (box.Height(10) ; box.Position([20*i,20*j,5]) )
            )
        )
    newScene.View() ;

```

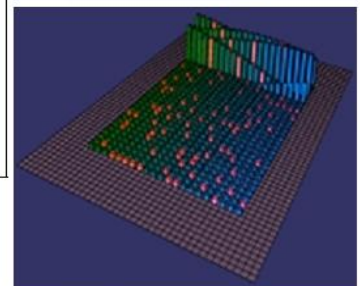
Listing 7: ZLang code demonstrating a key-framed animation; in which a series of boxes increase and decrease their height at a specific sequence.



Frame 15



Frame 220



Frame 350

every edge and draw it separately. Listing 6 and Listing 7 show examples for ZLang code utilizing the DCC engine.

5. IMPLEMENTATION

Performance considerations and tools availability were primary reason for choosing C++ as an implementation language. In choosing tools for the system, we focused on high performance, open source or free, cross platform tools with large communities. Modular reusable design was a main goal. Every module in the DCC Engine compiles to an independent shared library, and those shared libraries are then used from the plug-in code that glues the Engine and the interpreter.

We implemented a cross-platform interpreter for ZLang, where we built a scanner for ZLang that feeds a parser that generates an Abstract Syntax Tree (AST). A tree walker interprets each statement in the AST individually and executes it. Our type-system implementation makes heavy use of C++ templates and policy-based design [48], all internal data types are used as policies that sculpt specific functionalities of a general object, and these general objects are grouped in ZTvar. For the DCC engine, we implemented algorithms for the construction of primitives, application of modifiers, and connecting geometric primitives to a chosen physics and texture primitive. We also implemented a system for managing both types of animation (key-framed and physics). Each primitive added to scene can have only one of these animation types applied to it. At each frame, all added primitives are enumerated and the system delegates to the primitive, applying effect of its respective animation type and then draws the scene. Figure 9 shows the system architecture of ZLang.

For the DCC Engine, we used CGAL Polyhedron [49] (that acts as a high level interface to CGAL's half-edge datastructure [50]) for storing meshes. We used OpenSceneGraph[51], to manage the scene graph, draw our objects, and apply special effects to the scene (colors, lights, and textures). We used PhysX [9], as a physics engine (we preferred it over Bullet [52] due to its current GPU acceleration). We used Eigen [53], for efficient, complex matrix manipulations. We used GSL [54], for interpolating values for key-framed animation using cubic splines. We used GMP [55], to maintain precision of predicates in CGAL [56]. We used OpenNL[57], for solving differential equations associated with general method of texture mapping objects [58]. We used BOOST BGL [59], for a number of graph algorithms used by the geometric simplification algorithm.

For the interpreter, we used ANTLR [60], as lexer and parser generator. We used Google Sparse Hash [61], as a very efficient hash table used for implementing the symbol table. We used BOOST Variant [62], as an efficient, templated, and well managed union type that simplified generalizing the interpreter's source code. We used Boehm GC [63], for managing memory and efficiently reclaiming unreferenced memory locations during system operation.

6. CONCLUSION AND FUTURE WORK

We have presented ZLang, a scripting language for DCC applications. ZLang is general purpose, imperative, dynamic-strongly typed, hybrid paradigm and memory managed language. ZLang can be interpreted, compiled, extended, and embedded in C/C++. ZLang interpreter is cross-platform, free, and open source. We have demonstrated the reasoning behind each of these properties of ZLang, and how they are used together to fill holes that existed in previous systems.

ZLang interacts with a DCC engine through its standard library. The DCC engine provides geometric primitives, mesh modifiers, key-framed animation, physics simulations, and visualization effects (Texture and lighting). We have discussed each of these capabilities and provided one sample on construction of each of them. We provided ZLang examples that demonstrate some of its DCC features and show its general purpose abilities. ZLang is available online [64].

There are many interesting avenues for future work. First, we are investigating methods of integrating a GPU pipeline to ZLang, in which meshes are stored in the GPU, modifiers implemented as OpenCL Kernels [65] applied to them directly on GPU, and then they are drawn from the GPU with no need for inter CPU ↔ GPU data transfer. Second, a friendly GUI can be built on top of the DCC engine similar to one existing in 3Ds Max, Maya, and Blender. This will give a smoother experience to the artists during prototyping. Third, currently, only an interpreter is implemented for ZLang, a compiler can be built and will allow for faster execution time, and for building standalone simulations that doesn't depend on presence of the interpreter. We are intending to carry on the development of ZLang using the open source model.

7. REFERENCES

- [1] J. K. Hahn, Realistic animation of rigid bodies, SIGGRAPH Comput. Graph. 22 (1988) 299-308.
- [2] W. T. Reeves, Particle systems a technique for modeling a class of fuzzy objects, ACM Trans. Graph. 2 (1983) 91-108.
- [3] J. K. Ousterhout, Scripting: Higher-level programming for the 21st century, Computer 31 (1998) 23-30.
- [4] CGWiki, Maxscript (Apr. 2008).
<http://wiki.cgsociety.org/index.php/MAXScript>
- [5] M. R. Wilkins, C. Kazmier, MEL Scripting for Maya Animators, Second Edition (The Morgan Kaufmann Series in Computer Graphics), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [6] M. Tigges, B. Wyvill, Python for scene and model description for computer graphics, Proc. IPC 2000.
- [7] C. W. Reynolds, Computer animation with scripts and actors, in: Proceedings of the 9th annual conference on Computer graphics and interactive techniques SIGGRAPH '82, ACM, New York, NY, USA, 1982, pp. 289-296.
- [8] S. F. May, W. E. Carlson, F. Phillips, F. Scheepers, AI: a language for procedural modeling and animation, Technical report OSU-ACCAD-12/96-TR5, The Ohio State University CSIR (1996).
- [9] Nvidia, Physx (2011).
<http://www.geforce.com/Hardware/Technologies/physx>
- [10] Autodesk, Maya (2011).
<http://usa.autodesk.com/maya/>
- [11] Autodesk, 3ds max (2011).
<http://usa.autodesk.com/3ds-max/>
- [12] M. Lutz, Programming python, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.
- [13] Blender Foundation, Blender (2011).
<http://www.blender.org>

- [14] Aristid, Lindenmayer, Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs, *Journal of Theoretical Biology* 18 (3) (1968) 300 - 315.
- [15] P. Prusinkiewicz, A. Lindenmayer, *The algorithmic beauty of plants*, Springer-Verlag New York, Inc., New York, NY, USA, 1996.
- [16] Y. I. H. Parish, P. Müller, Procedural modeling of cities, in: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques, SIGGRAPH '01*, ACM, New York, NY, USA, 2001, pp. 301-308.
- [17] G. Stiny, *Pictorial and Formal Aspects of Shape and Shape Grammars*, BirkhauserVerlag, Basel, Switzerland, 1975.
- [18] P. Wonka, M. Wimmer, F. Sillion, W. Ribarsky, Instant architecture, *ACM Trans. Graph.* 22 (2003) 669-677.
- [19] P. Müller, P. Wonka, S. Haegler, A. Ulmer, L. Van Gool, Procedural modeling of buildings, *ACM Trans. Graph.* 25 (2006) 614-623.
- [20] J. M. Snyder, Generative modeling for computer graphics and CAD: symbolic shape design using interval analysis, Academic Press Professional, Inc., San Diego, CA, USA, 1992.
- [21] S. Havemann, D. W. Fellner, Generative mesh modeling, Ph.D. thesis (2005).
- [22] R. Cartwright, V. Adzhiev, A. A. Pasko, Y. Goto, T. L. Kunii, Web-based shape modeling with hyperfun, *IEEE Comput. Graph. Appl.* 25 (2005) 60-69.
- [23] B. Cutler, J. Dorsey, L. McMillan, M. Müller, R. Jagnow, A procedural approach to authoring solid models, *ACM Trans. Graph.* 21 (2002) 302-311.
- [24] W. T. Reeves, E. F. Ostby, S. J. Leffler, The menumodelling and animation environment, *The Journal of Visualization and Computer Animation* 1 (1) (1990) 33-40.
- [25] S. Upstill, *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
- [26] K. Perlin, E. M. Hoffert, Hypertexture, *SIGGRAPH Comput. Graph.* 23 (1989) 253-262.
- [27] L. Velho, K. Perlin, L. Ying, H. Biermann, Procedural shape synthesis on subdivision surfaces, in: *Proceedings of the XIV Brazilian Symposium on Computer Graphics and Image Processing, SIBGRAPI '01*, IEEE Computer Society, Washington, DC, USA, 2001, pp. 146-153.
- [28] F. K. Musgrave, C. E. Kolb, R. S. Mace, The synthesis and rendering of eroded fractal terrains, in: *Proceedings of the 16th annual conference on Computer graphics and interactive techniques, SIGGRAPH '89*, ACM, New York, NY, USA, 1989, pp. 41-50.
- [29] R. Szeliski, D. Tonnesen, Surface modeling with oriented particle systems, *SIGGRAPH Comput. Graph.* 26 (1992) 185-194.
- [30] T. Lewis, M. W. Jones, A system for the non-linear modelling of deformable procedural shapes, *The Journal of WSCG* 12 (2) (2004) 253-260.
- [31] K. Perlin, A. Goldberg, Improv: a system for scripting interactive actors in virtual worlds, in: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, SIGGRAPH '96*, ACM, New York, NY, USA, 1996, pp. 205-216.
- [32] K. Perlin, Real time responsive animation with personality, *IEEE Transactions on Visualization and Computer Graphics* 1 (1995) 5-15.
- [33] C. Elliott, Modeling interactive 3d and multimedia animation with an embedded language, in: *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL)*, 1997, USENIX Association, Berkeley, CA, USA, 1997, pp. 22-22.
- [34] J. Cohen, Garbage collection of linked data structures, *ACM Comput. Surv.* 13 (1981) 341-367.
- [35] F. Henglein, Dynamic typing, in: B. Krieg-Bräse, (Ed.), *ESOP '92*, Vol. 582 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 1992, pp. 233-253.
- [36] K. Arnold, J. Gosling, *The Java programming language* (2nd ed.), ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
- [37] M. Mantyla, *An introduction to solid modeling*, Computer Science Press, Inc., New York, NY, USA, 1987.
- [38] H. Bendels, D. W. Fellner, S. Havemann, Modellierung der grundlagen - erweiterbare datenstrukturen zur modellierung und visualisierung polygonaler welten, in: *Modeling - Virtual Worlds - Distributed Graphics*, infix, 1995, pp. 149-158.
- [39] A. H. Barr, Global and local deformations of solid primitives, *SIGGRAPH Comput. Graph.* 18 (1984) 21-30.
- [40] E. E. Catmull, A subdivision algorithm for computer display of curved surfaces., Ph.D. thesis (1974).
- [41] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, W. Stuetzle, Multiresolution analysis of arbitrary meshes, in: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques, SIGGRAPH '95*, ACM, New York, NY, USA, 1995, pp. 173-182.
- [42] E. Welzl, Smallest enclosing disks (balls and ellipsoids), in: H. Maurer (Ed.), *New Results and New Trends in Computer Science*, Vol. 555 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 1991, pp. 359-370.
- [43] P. Alliez, S. Pion, Fast principal component analysis, in: C. E. Board (Ed.), *CGAL User and Reference Manual*, 3rd Edition, 2007.
- [44] C. B. Barber, D. P. Dobkin, H. Huhdanpää, The quickhull algorithm for convex hulls, *ACM Trans. Math. Softw.* 22 (1996) 469-483.
- [45] P. Lindstrom, G. Turk, Fast and memory efficient polygonal simplification, in: *Proceedings of the conference on Visualization '98, VIS '98*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1998, pp. 279-286.
- [46] P. Lindstrom, G. Turk, Evaluation of memoryless simplification, *IEEE Transactions on Visualization and Computer Graphics* 5 (1999) 98-115.
- [47] J.-M. Lien, N. M. Amato, Approximate convex decomposition of polyhedra, in: *Proceedings of the 2007*

- ACM symposium on Solid and physical modeling, SPM '07, ACM, New York, NY, USA, 2007, pp. 121-131.
- [48] A. Alexandrescu, Modern C++ design: generic programming and design patterns applied, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [49] L. Kettner, 3d polyhedral surfaces, in: C. E. Board (Ed.), CGAL User and Reference Manual, 3rd Edition, 2007.
- [50] L. Kettner, Halfedge data structures, in: C. E. Board (Ed.), CGAL User and Reference Manual, 3rd Edition, 2007.
- [51] D. Burns, R. Osfield, Openscenegraph (osg) (2011).
<http://www.openscenegraph.org>
- [52] E. Coumans, et al., Bullet (2011).
<http://bulletphysics.org>
- [53] B. Jacob, Eigen (2011).
<http://eigen.tuxfamily.org>
- [54] GNU, Gnu scientific library (gsl) (2011).
<http://www.gnu.org/software/gsl>
- [55] GNU, Gnu multiple precision arithmetic library (gmp) (2011).
<http://gmplib.org>
- [56] H. Brnimmann, A. Fabri, G.-J. Giezeman, S. Hert, M. Ho mann, L. Kettner, S. Schirra, S. Pion, 2d and 3d geometry kernel, in: C. E. Board (Ed.), CGAL User and Reference Manual, 3rd Edition, 2007.

- [57] INRIA, Open numerical library (opennl) (2010).
<http://alice.loria.fr/index.php/software/4-library/23-opennl.html>
- [58] L. Saboret, P. Alliez, B. LA(c)vy, Planar parameterization of triangulated surface meshes, in: C. E. Board (Ed.), CGAL User and Reference Manual, 3rd Edition, 2007.
- [59] J. Siek, L.-Q. Lee, A. Lumsdaine, Boost graph library (bgl) (2011).
URL http://www.boost.org/doc/libs/1_47_0/libs/graph
- [60] T. J. Parr, R. W. Quong, Antlr: a predicated-ll(k) parser generator, Softw. Pract. Exper. 25 (1995) 789-810.
- [61] Google, google-sparsehash (2011).
<http://code.google.com/p/google-sparsehash>
- [62] E. Friedman, I. Maman, Boost variant (2011).
http://www.boost.org/doc/libs/1_47_0/doc/html/variant.html
- [63] H.-J. Boehm, M. Weiser, Garbage collection in an uncooperative environment, Softw. Pract. Exper. 18 (1988) 807-820.
- [64] M. Yousef, A. Hashem, H. Saad, K. F. Hussain, Zlang (2011).
<http://z-lang.sourceforge.net>
- [65] A. Munshi, Opencl: Parallel computing on the gpu and cpu, ACM SIGGRAPH Tutorial (2008).

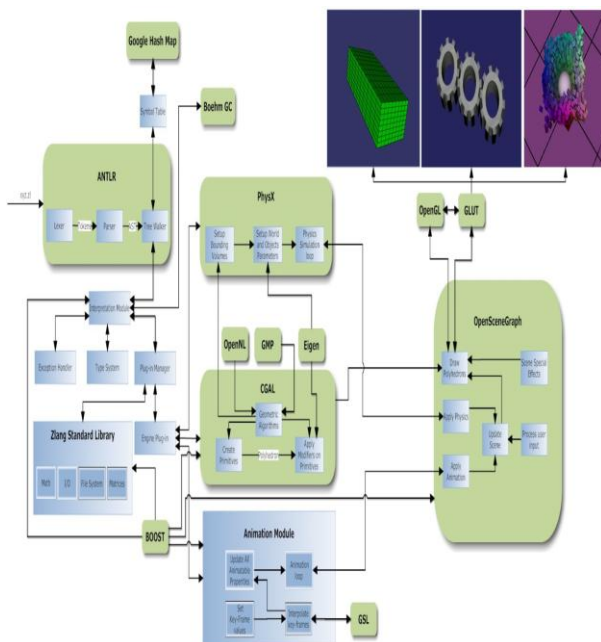


Figure 9:ZLang Architecture. The diagram illustrates the various components that collaborate together to execute a ZLang script, each module is named by the main library used to construct its components. The arrows illustrate inter-module and intra-module collaborations. A typical execution that utilizes the graphics engine goes from left to right starting from an input file xyz.zl, till it's outputted to user in one of three forms; a still model, a key-frame animation , or a physics simulation.