MDSD for Games with Eclipse Modeling Technologies

Steve Robenalt

MDSD for Games with Eclipse Modeling Technologies

S. A. Robenalt

Web Circuit, Cameron Park, CA 95682, USA
steve@webcircuit.com

Abstract. This paper describes an approach to be used to apply Model Driven Software Development (MDSD) techniques to the development of software for 3D games on multiple platforms, with additional usage in Scientific Visualization, Virtual Reality (VR), Augmented Reality (AR), and Animation applications. Specific areas of applicability for MDSD techniques include development tooling and artifact creation. The MDSD support for this functionality can be implemented as extensions to the Eclipse Platform augmented by a subset of the modeling projects that are available with Eclipse.

1 Introduction

The development of 3D game software presents some unique technical challenges when compared to other types of software. Principal among these challenges is the desirability of supporting multiple platforms with minimal effort and maximal (platform specific) performance. Supporting multiple platforms typically implies not only multiple programming languages (C, Objective-C, C++, Java, etc.), but also different graphics libraries (OpenGL, DirectX, OpenGL ES), different versions of those libraries (OpenGL 2.X, 3.X, and 4.X), and extensions that introduce features from later versions in conjunction with an earlier API version. Maximizing performance generally involves optimizing source code structure for specific target platforms, as well as taking advantage of newer graphics APIs or API extensions when they are available. In general, the selection of programming language has a lesser impact on performance when compared to the graphics libraries since the latter achieves performance improvements by exploiting hardware dedicated to the purpose (e.g. GPUs).

In practice, the choice of both programming language and graphics library are generally dictated by the hardware vendors and the supported operating systems for the target platforms.

The content of this paper is based heavily on the author's experience in developing and maintaining an Eclipse-based MDSD tool chain in an enterprise software environment, and on recent ongoing work to build Eclipse-based MDSD tooling for a new game engine architecture.

2 Existing Strategies

The conventional solution for supporting multiple platforms for game development involves the selection and modification (or less often, creation) of a game engine - a set of related software artifacts that have already been ported to multiple platforms - followed by customization of the result to include content based on the theme of the game under development. One of the earliest examples of a game engine was the Quake engine, used in the game of the same name. Other game engines are available as well, such as the Unreal engine, and the Unity 3D engine, which are both proprietary, and which provide a complete development environment and runtime for games that can be targeted to multiple platforms.

Cross platform support in these engines can be implemented using a variety of mechanisms, most of which are based on segregation of code into platform independent layers and platform specific layers, then using a combination of conditional compilation and conditional build steps to handle the targets.

Supporting multiple graphics (or other) libraries can use the same conditional compile/build facilities, or in more extreme cases, can cause the need to create separate but functionally equivalent (from the perspective of the game) versions of the same software, which can be maintained in parallel.

Alternatively, games can be implemented using a Virtual Machine approach with a common graphics API (e.g. Java Virtual Machine + Lightweight Java Game Library), in which case the game-specific code is platform neutral and the JVM and game library handle the platform specific differences. Note that in this case, the game library does not insulate the developer from handling multiple versions of the underlying OpenGL implementation, which is usually tied to the hardware and device drivers of the graphics adapter.

3 Model Driven Strategies

While the above methods for supporting multiple platforms have all been used over many years, and are successful to varying degrees, a newer approach involves the introduction of Model Driven Software Development (MDSD) techniques into the development process.

3.1 Idealized MDSD

In the idealized MDSD case, a single Platform Independent Model (PIM) of the game is created and maintained. For each target platform, the PIM is transformed into a Platform Specific Model (PSM), which reflects the constraints and capabilities of the target platform, then the PSM is transformed into source code that is specific to (and more optimized for) that particular target.[1]

3.2 Variations

There are many practical variations of this idealized case:

1. The transformation to PSM is omitted, and the required transformations are implemented in the source code templates.
2. The transformation from PSM to source code is omitted, and platform specific object code is produced directly from the PSM.
3. The PSM is interpreted directly by the game runtime, rather than compiled.
4. The transformation from PIM to PSM may also have multiple stages.

The use of these variations depends on a balance of complexity in the process relative to complexity in the tool chain (e.g. code templates). These variations generally do not impact the resulting generated code, but they do impact the creation and maintenance of the components of the MDSD tooling.

3.3 Advantages

The major advantages of the idealized case are:

1. The PIM becomes the single, definitive reference for all versions of the software.
2. The PIM is substantially simpler and more comprehensible than the source code it will eventually produce, thus it is quicker to create and easier to maintain.
3. The PSM provides an intermediate checkpoint where platform constraints can be validated and enforced.
4. The source code can also be audited to insure that it is complete, correct, and efficient.
5. The source code also provides the ability to debug code that contains errors, which can be traced back to the code templates.
6. Code consistency and quality improve quickly based on the use of few templates to generate many artifacts.

3.4 Disadvantages

The major disadvantages of the idealized case are:

1. A practical MDSD tool chain requires a consistent architecture in the PIM specification, and significant expertise in creating the code templates (i.e. most experienced personnel).
2. In practice, the PIM is often an incomplete model, which leaves parts of the application unspecified.
3. Compensating for an incomplete PIM generally means integrating generated and handwritten code, which requires care and planning.
4. Coordinating handwritten and generated code complicates the evolution of the software over time.

Compensating for these disadvantages can eliminate or substantially reduce their impact, but often, the insight needed to develop compensation only comes as a result of experience.

While the Idealized MDSD scenario has great potential, a more pragmatic approach is often appropriate. This approach involves applying MDSD selectively to parts of the development process. The artifacts that are selectively generated are then integrated into a standard development process involving code written by hand. This allows for selective introduction of MDSD into an existing development environment.

4 Using Eclipse for MDSD

Building an MDSD tool chain from scratch would be a very complex task. However, starting from existing tools which can be easily customized is much more feasible. The Eclipse Integrated Development Environment provides extensive support for MDSD tooling via numerous modeling related projects. Foremost among these is the Eclipse Modeling Framework [2], which provides the metamodel (Ecore) for many of the other projects, as well as for numerous non-modeling projects.

Some of the other relevant Eclipse modeling projects include:

1. CDO Model Repository - persistence and sharing of models.[1]
2. ATL - Model-to-model transformation (e.g. PIM to PSM).[2]
3. Xpand/Xtend - Model-to-text language for source code templates.[3]
4. Xtext - Domain Specific Language creation toolkit.[4]
5. Modeling Workflow Engine - Coordinated execution of the MDSD tool chain.[5]
6. OCL - Declarative specification of model constraints.[6]

By combining these projects, and optionally using some of the other related projects, a sophisticated MDSD tool chain for game development can be produced in a comparatively short period of time. In particular, the Xtext project provides good integration of text-based models into Eclipse itself, leveraging the support provided by the IDE when editing models and building artifacts from them.

Note that Eclipse has utility in the game development community outside of the above noted Modeling projects. Eclipse was created originally to integrate the diverse sets of tools used in software development into a common environment using a plug-in based architecture that simplifies integration. The same approach is feasible when applied to the many artifacts needed to create games.

---

[1] http://www.eclipse.org/cdo/
[2] http://www.eclipse.org/atl/
[3] http://www.eclipse.org/xtend/
[4] http://www.eclipse.org/Xtext/
[5] http://www.eclipse.org/projects/project.php?id=modeling.emf.mwe
[6] http://www.eclipse.org/modeling/mdt/?project=ocl

5 Applying MDSD Techniques

The next section describes how MDSD techniques and the Eclipse modeling projects can be combined and used in various parts of the game development cycle.

5.1 Development Tooling

The development of games requires the production and coordination of a large number of artifacts, both in the form of descriptive elements representing the game environment, and in the production of source code that renders the environment, allows player interactions, handles the progress of play, and renders the scene in the environment frequently enough to provide smooth animation.

Environment Creation A large number of descriptive elements are used in the construction of game environments. The following are some major classes of these elements:

1. Geometry - Mathematical descriptions of the surfaces of the environment, players, and objects comprising the game.
2. Textures - Descriptions of surface textures, either through stored or dynamically created images projected onto the geometry.
3. Attributes - Details (e.g. material properties) added to the geometric elements to enhance the realism of the environment.
4. Lighting - Location and nature of light sources which affect the visible properties of the elements of the environment.
5. Animation - Definitions of how elements of the environment change over time, both independently, and in response to player actions.
6. Sounds - Used to enhance the realism of the environment; generally tied together with animations or player actions.

Development tooling can be provided to support the creation and maintenance of each of these classes of elements, while MDSD tooling can be used to project these elements to the target environment. For example, the PIM for surface geometry would store surface descriptions using double precision floating point numbers, while the transformation to the PSM and then source code would reduce the surface description to single precision or fixed point numbers based on the capabilities of the target.

Shader Programs One of the major changes that has been driving improvements in the performance and realism of games for the last several years has been the introduction of programmable hardware, along with the development of Domain Specific Languages (DSLs)[3] that are used to write the programs. For historical reasons, these languages are commonly called shader languages, named for type of artifact they were originally used to create. More recently,

these programs have been used to create animation effects, and even to handle more general workloads such as physics calculations.

One of the most common DSLs used for this purpose is GLSL[4],[5], which is part of the OpenGL specifications[6],[7]. Based on the need to keep up with performance improvements made possible by hardware enhancements, and to reflect the growing diversity of computational tasks that are being delegated to the GPU, the GLSL language itself has changed substantially in recent years. Also, the GLSL language has variants that are specific to specific parts of the programmable hardware.

MDSD techniques can be used for tooling support for shader programs in two distinct areas. The first area involves using the Xtext DSL toolkit to add support to the Eclipse IDE for GLSL as a language. This provides for productivity enhancements such as syntax highlighting in editors and code completion suggestions. The second area involves modeling the shader progams using a PIM, then adding PSMs and source templates to target the shader programs to multiple versions, or even to alternative shader languages.

5.2 Artifact Creation

One of the primary uses for MDSD techniques involves the ability to support multiple target platforms from the same PIM. Depending on the diversity of target platforms to be supported, source code artifacts must be created to account for many variations:

1. OS/Window System - Windows, OSX, Linux, IOS, Android, Firefox, Chrome
2. Platform Type - Desktop, Tablet, Phone, Browser
3. Programming Language - C, C++, Java, Objective-C
4. API Version - OpenGL 2.X, 3.X, 4.X, OpenGL ES 2.X, WebGL, DirectX

Creating these artifacts involves developing model-to-model transformations for the PIM to PSM phase, and model-to-text templates for the source code creation phase. The number and complexity of the transformations and templates can vary based on the valid combinations of the above factors, but in general, is a small fraction of the number of actual artifacts that are generated.

6 Summary

There are many areas of game software development where MDSD techniques can be exploited to streamline the development process and improve code consistency and quality. The Eclipse IDE and Modeling frameworks provide a solid foundation upon which a game-specific MDSD tool chain can be built, and the resulting tools can be integrated into Eclipse itself to augment existing development processes.

Since the application of MDSD tools in other software disciplines (notably enterprise and mobile) has yielded substantial benefits, there is certainly a strong motivation to pursue MDSD in game development as well.

References

1. A. Kleppe, J. Warmer, W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise.* Addison Wesley, 2003, pp. 5–8.
2. D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF Eclipse Modeling Framework. Second Edition* Addison Wesley, 2009.
3. M. Fowler, R. Parsons *Domain-Specific Languages* Addison Wesley, 2010
4. http://www.opengl.org/documentation/glsl/
5. R.J. Rost, B. Licea-Kane, et al. *OpenGL Shading Language* Addison Wesley, 2010.
6. http://www.opengl.org/documentation/
7. D. Shreiner. *OpenGL Programming Guide. Seventh Edition* Addison Wesley, 2010.