

# Incorporating “Level Of Detail” into Programming Languages Aimed at Real-Time Application

Takahito Tejima

Yasushi Kuno \*

2006.1.18

## Abstract

Interactive computer graphics, including video games, are now popular and widely adopted computer usage. However, detailed scene generation consumes lot of CPU power and thus might prohibit timely update of graphics views. To address the problem, some programs uses “Level Of Details” (LODs) to control the conciseness of scene being generated; use lager LODs when there is enough CPU power (spare processing time), and use smaller LODs when not. Currently such programs are written in traditional programming languages, and LOD mechanisms are hand-crafted by the programmes, which is tedious and error-prone. In this paper, we propose alternative method in which LODs are directly supported by programming language core. With our approach, execution of alternative code corresponding to several levels of LODs are automatically controlled and selected by the language runtime, leading to more readable and maintainable code. Core framework of proposed language mechanism along with experimental language specification and implementation plans are presented.

## 1 Introduction

Real-time processing is a processing method that is widely used in interactive graphics applications such as in video games and visualization simulation as well as in built-in systems such as computer controls and communication devices.

Generally, processing accuracy and interactivity are in a trade-off relation. In video games and visualization simulation, a widely used method is to prepare data regarding the objects that are to be displayed at multiple detail levels, and use the data while changing such levels depending on the priority of factors. This method is known as *level of detail* or *LOD* for short [5][3](Figure 1).

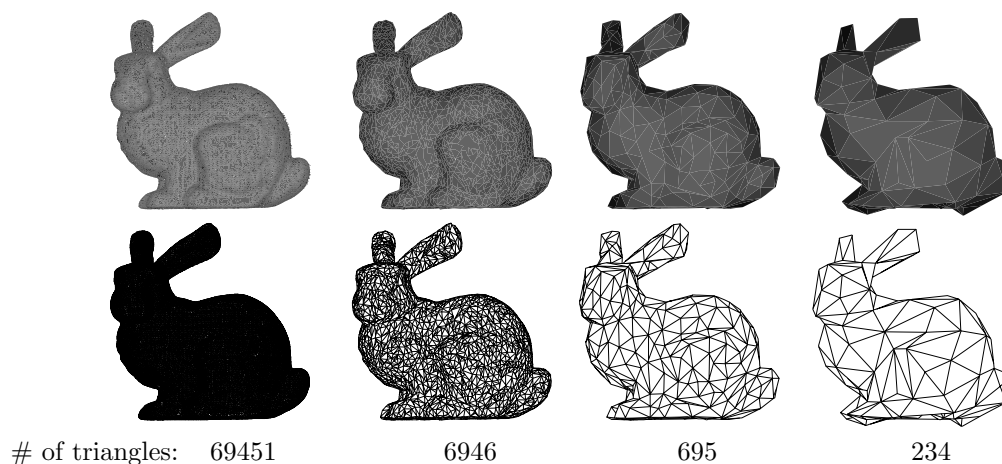


Figure 1: Shape-LOD examples

For example, objects viewed in the distance do not need detailed shapes since they are displayed in a small size on the screen. In this case, utilizing a shape with a low detail level (rough shape) can increase throughput.

This can also be applied to, not only the display of contents, but also the processing of behaviors and actions. As an example, when stadium spectators are to be drawn, choices as shown in Table 1. are possible.

---

\*Graduate School of Systems Management, The University of Tsukuba, Tokyo




Accuracy	High	Medium	Low
Cost	High	Medium	Low
Processing examples	Simulate gaze conditions and corresponding body movement per individual	Complete control of behaviors corresponding to outside events per group	Play predetermined animations in all spectator seats.
			

Table 1: LOD example in spectator behavior

Similarly, when displaying walking people, the choices as shown in Table 2. are possible.

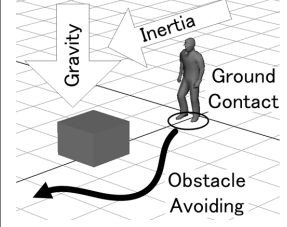
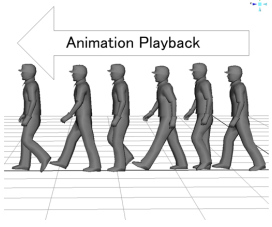
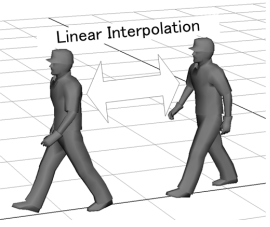
Accuracy	High	Medium	Low
Cost	High	Medium	Low
Processing examples	Dynamically determine each hand and foot movement while calculating gravity, inertia, and contact with the ground	Play walking animations prepared for the hand and foot bone structure in advance	Display a shape with a right foot forward and that with a left foot forward by linear interpolation
			

Table 2: LOD example of walking people

In real-time graphics applications, the task is cyclic as shown below:

```

loop {
  advance current-time by small amount
  generate scene according to the current time
}

```

To generate a scene, controlling (1) states of the each objects to be displayed, (2) LODs for each of those objects, and (3) set of the objects currently visible on the screen are necessary. As conventional programming languages do not support above controls directly, we have to code them manually, as in the following outline.

```

loop {
  advance current-time by small amount
  for each object o within the scene{
    if(o is visible on the screen){
      if(state of o is X){
        if(LOD for o is high){
          /* high resolution drawing of o in state X */
        }else if(LOD for o is moderate){
          /* middle resolution drawing of o in state X */
        }
        ...
      }
    }
  }
}

```

```

}else if(state of o is Y){
    if(LOD for o is high){
        ...
        ...
    }
}
}
}
}
}
}

```

Such coding is tedious, hard to read and error-prone. To overcome the problem, this report proposes an object oriented language that supports processing depending on the detail level. First, we will describe the outline of time restriction in graphics applications and LOD processing of data from previous studies, explaining the necessity for process dependent processing according to detail level. Then, real-time languages in previous studies will be examined. A programming language framework that enable concise description of automated LOD controls for processes will be discussed.

## 2 Time Restriction and LOD Processing

Interactive graphics applications are categorized as either variable frame rate systems or fixed frame rate systems, depending on the time it takes to update output images[3].

In a variable frame rate system, a drawing process is performed in a time proportional to the number or complexity of objects to be drawn. However in most cases, since the complexity of the objects to be drawn greatly depends on the situation such as the state of each object or viewpoint coordinate. The problem of variable frame rate system is that a stable real-time characteristic cannot be obtained[6].

Conversely, in a fixed frame rate system, the drawing process on the screen is performed in synchronous to the refresh rate of the output equipment. For example, since the output equipment in non-portable household game machines is generally the TV monitor, the drawing process is performed as a cyclic task conforming to video signal specifications such as NTSC (60 cycles/second) and PAL (50 cycles/second). Because interactive applications require not only drawing processing but also simultaneous game-model simulation according to the user input, scenario progress, and sound processing, CPU time that can be used for the drawing processing is much more restricted.

In both variable and fixed systems, a LOD processing that has prepared hierarchical data on drawn objects and uses prioritized simplified data has been used for a long time[5]. Figure 2 shows timing change and real-time characteristic improvement by applying LOD. The time required for the drawing process of A, B, and C models per frame is shown in the upper half of Frame 1. Drawing all of A, B, and C as is, will exceed the deadline and prevent updating of drawing contents in the next frame. Here, when B's importance is low, by switching simple model B' instead of B on the screen can reduce the total cost, allowing a new drawing process in the next frame as shown in the lower half of the diagram. On the other hand, when the importance of B is high, switching to models A' and C' for A and C can adjust the load as shown in the lower half of Frame 3.

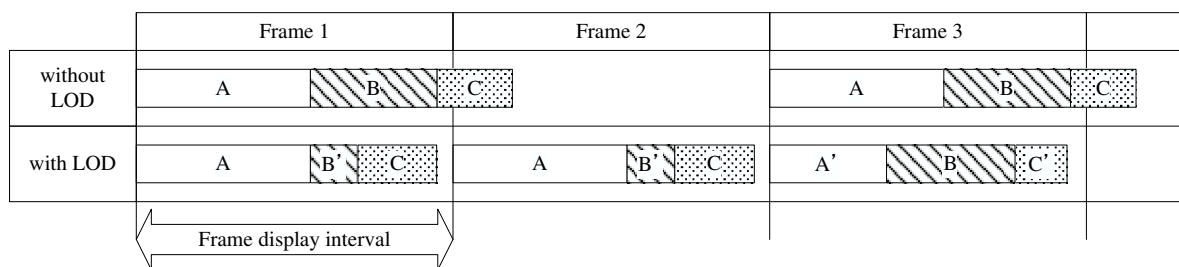


Figure 2: Changing Real-Time Characteristics With or Without LOD Processing

Many studies have been conducted regarding methods to simplify geometrical data[3]. Method to decide which detail level should be used at implementation greatly differs between a variable system and a fixed system.

Detail levels are selected by individual data in a variable system, while in fixed systems, they are selected by a scheduler considering the total throughput and time restriction in a fixed system. Of course, a fixed system has greater real-time characteristics.

Thus, adaptive detail level selection of graphical data has been widely applied.

On the other hand, recent improvements in hardware performance has not only greatly expanded the amount of processing graphical data but also the complexity of processing methods. Physical simulation is used for describing object behaviors in real-time graphics applications as well, and for crowd expression, processes responding to a variety of events in the scene can be written in a program[10]. LOD processing can be similarly applied to such behavior describing, and substituting omitted behavior processing for objects with low importance on the screen because they are viewed in the distance can reduce the total throughput. However, the problem lies in that in most cases, a detail level switch process must be described in conjunction with the behavior process, and current lack of descriptive power in the conventional programming language makes it difficult to completely ensure time restriction.

### 3 Language Support for Real-Time Processing

Schedule systems for real-time processing have been widely studied. RTC++[9] is an extension of C++ and its features are:

- a real-time object which is an active entity
- creation of an active entity on a remote host
- timing constraints in an operation as well as in statements
- a periodic task with rigid timing constraints

The system is based on an object model, and attempts to add a time restriction by a method or statement and make the processing time a module in a distributed environment. By using the time restriction specified in process code, scheduling of a task comprised of plural object calls and coding of exceptional processes when exceeding the deadline are possible.

Such a framework is also applicable in a real-time visualization applications which have specific characteristics that are not found in standard built-in systems. The point is that when performing processes cannot meet the time restriction, an alternative measure using lower quality is used. This is a very important characteristic in real-time processing, but general languages do no support alternative measure selection and implementers perform such selection on an individual problem basis under the current circumstances. Some libraries[1] can perform such selection, but they must use scene graph structures specific to the libraries and are not suitable for use when extracting only a process describing.

## 4 Level of Detail Processing System

### 4.1 Language Design

Based on the above analysis and application examples, we propose a programming language which enables flexible and easy coding with strict time restrictions. This study examines its specifications and experimental implementation to evaluate its usability by some execution examples.

In consideration of an easy-to-use integration to current applications, the language proposed in this study was configured as an expanded implementation method of C++ classes. Therefore, the detailed grammar follows C++, and the processing system ultimately generates C++ source codes.

The following is class A declaration with method `f()`.

```
/* -- Declaration -- */
class A {
public:
    void f();
};
```

Method `f()` has multiple implementation, each of which is appropriate for the some specific range of LODs. In our design, LODs are controlled by a small integer called “cost”. The cost value is interpreted as the processing time required to run the method. Each implementation of `f` is accompanied by a cost value as in the form `f{cost}(...)`. Any number of `f()` can be defined, provided that their cost value all differ each other.

```
/* -- Implementation 1 -- */
A::f{100}()
{
    <Highly accurate processing that requires cost>
```

```

}
/* -- Implementation 2 -- */
A::f{50}()
{
  <Coarse processing available at half the cost>
}

```

On the user side of this class object, schedule block as in the following form is used:

```

schedule(limit){
  code to be executed
}

```

Here is code example that execute a.f(), b.f() and c.g() with total cost limit 200. The meaning of **priority** will be explained later.

```

/* -- Using objects -- */
{
  A a, b;
  C c;

  a.priority = 5;
  b.priority = 3;
  c.priority = 1;

  schedule(200){
    a.f();
    b.f();
    c.g();
  }
}

```

Above is the basic outline of this language. Which implementation of `f()` is actually used is determined dynamically. Giving method execution time to consumed costs and putting a time restriction on the schedule limit value can achieve real-time processing.

In the current design, no flow controls such as loop within a schedule block are allowed.

## 4.2 Implementation Plan

A selection plan of shape data detail levels in real-time graphic applications has two types of methods: a “reactive scheduler” and “predictive scheduler”. The graphics application is a cyclic task implementation as described above, and the correlation between frames is very high. A reactive scheduler is a method that changes the current selection depending on whether the results of the detail level selection of the previous frame exceeds the cost restriction or not[2][7][8]. This method is effective if processing costs between frames does not change largely[1], but it is problematic when frame correlation cannot be used.

On the other hand, predictive scheduler estimates actual processing cost (by either programmer annotations or by actually running the target code) and use the estimated cost for scheduling choices. Thus, while requiring some fixed overhead, a predictive scheduler can be used when processing cost of each frames varies largely (as in non-cyclic tasks[4]). Since this study focuses, not on the shape selection, but on the processing selection, a predictive scheduler is adopted.

We explain our scheduling method using the above code fragment as an example. Inside of the scheduling block is processing in 2 paths.

In the preparation phase, all method invocation instances with multiple implementation are listed in the table, along with available cost choices (Table 3. In this example, two method invocation exists for the same method `f()`. So, available costs are same for both instances).

Then possible combination of choices are examined, and one that satisfies the specified limit is selected.

In the execution phase, selected implementation is called for each invocation instances. Actually there will be many combinations that satisfy specified time limit. The system selects among them examining priority (the right most column in the table).

Thus the programmer can control the system’s scheduling choices by changing the priority values for each invocation instances.

This optimization is a kind of knapsack problem and belongs NP-complete, but we need the fastest possible solution, not the most exact. Therefore, we have adopted a greedy algorithm that begins the lowest cost detail level of each method and increases the detail level of according to the selected methods.

instance	method	cost of impl.1	cost of impl.2	...	priority
a	f()	100	50	...	5
b	f()	100	50	...	3
c	g()	70	30	...	1

Table 3: Table of entries to be invoked

## 5 Discussion

Experimental implementation is in its initial stage and many necessary requirements have yet to be met.

- Whether there can be a method selection scale other than cost  
Cost: static, defined at compiling, depending on the method  
Priority: dynamic, changes with context, depending on the instance
- A function that selects a lower detail level for lower priority items is needed when multiple implementation methods have the same cost
- Similarly, a structure to control priority by instance in a specific execution context should be adopted.
- A method to write situation differences, such as a view hardly seen from the side (less important) but commonly seen from the top (more important) in scene graphs.
- Whether cost of an actual method based on detail level at compiling can be automatically calculated, or whether a structure to store and utilize cost performance measured at implementation can be created.
- Countermeasures for when a certain method calls another method inwardly, and both have multiple detail levels.
- A method to assure continuity when instance detail levels between frames change and the timing to initialize member variables that only use specific detail level implementation.

## 6 Summary and Conclusion

In this paper, we proposed programming language which support LODs. In contrast with manually programming of complex LOD branching, specialized language make it easier to separate LOD controls from other processing such as state controls. We have described basic design and implementation of such language.

## References

- [1] OpenGL Performer getting started guide. Technical Report 007-3560-002, Silicon Graphics, Inc., 2000.
- [2] J.M. Airey, J.H. Rohlf, and Frederick P. Brooks Jr. Towards image realism with interactive update rates in complex virtual building environments. *Proceedings of 1990 Symposium on Interactive 3D Graphics*, pages 41–50, 1990.
- [3] David Luebke et al. *Level of detail for 3D Graphics*. Morgan Kaufmann, San Francisco, 2003.
- [4] T.A. Funkhouser and C.H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Proceedings of SIGGRAPH93*, pages 247–254, 1993.
- [5] Clark J. H. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.
- [6] R Hawkes. Update rates and fidelity in virtual environments. *Virtual Reality: Research, Development, and Application*, 1(2):99–108, 1995.
- [7] L.E. Hitchner and M.W. McGreevy. Methods for user-based reduction of model complexity for virtual planetary exploration. *Proceedings of the SPIE, The International society for optical engineering*, 1913:622–636, 1993.
- [8] Holloway and R.L. Viper. A quasi-real-time virtual-worlds application. Technical Report TR-92-004, Department of Computer Science, University of North Carolina at Chapel Hill, 1991.

- [9] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer. An object-oriented real-time programming language. *Computer*, 25(10):66–73, 1992.
- [10] Daniel Thalmann Soraia Raupp Musse. Hierarchical model for real time simulation of virtual human crowds. *IEEE Transactions on Visualization and Computer Graphics*, 07(2):152–164, 2001.