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

Takahito Tejima and Yasushi Kuno ☆☆

Interactive computer graphics, including video games, are now popular and widely adopted computer usage. However, detailed scene generation consumes lots 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. We have implemented a experimental language based on our proposal. Also we present several examples written with the language.

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 choose the data of appropriate level considering the trade-offs noted above. This method is known as *level of detail* or *LOD* for short[2)3)] (Figure 1).

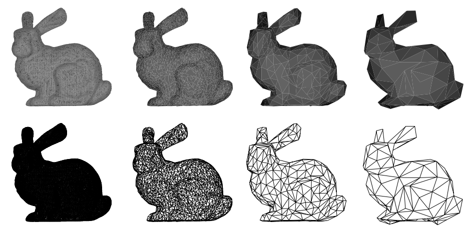


Fig. 1 Shape-LOD examples

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

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

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

```
loop {
  1. advance current-time by small amount.
  2. 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){
          ...
          ...
        }
      }
    }
  }
}
```

☆ Graduate School of Systems Management, The University of Tsukuba, Tokyo

Table 1 LOD example in spectator behavior

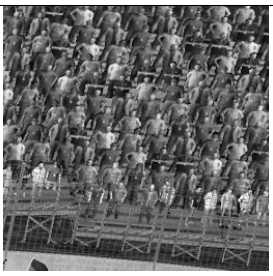
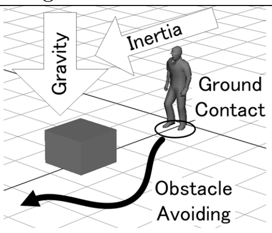
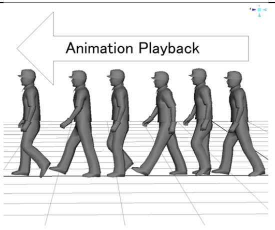
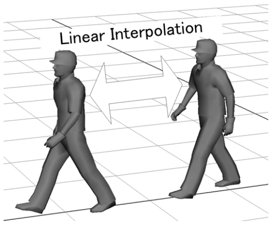
Accuracy	High	Medium	Low
Cost	High	Medium	Low
Processing examples	Simulate gaze conditions and corresponding body movement per individual	Complete control of behaviors corresponding to outside events per group	Play predetermined animations in all spectator seats.
			

Table 2 LOD example of walking people

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
			

Such coding is tedious, hard to read and error-prone. To overcome the problem, this paper proposes an object oriented language that supports automatic choice among alternative codes according to the required level of details.

First, we will describe the outline of time restriction in graphics applications and LOD processing of data from previous studies. We also explain the necessity for process dependent processing according to detail level. Then, real-time languages in previous studies will be examined. Finally, a programming language framework which enables description of multiple codes corresponding to multiple LODs and automatic choice among them 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 way they handle the timing constraints of updating images[2].

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 differs drastically depending 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[4].

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 other processing such as 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 frame-rate systems, a LOD processing based on hierarchical scene graph, which include multiple LOD choices, combined with simple priority scheme has been used for a long time[3]. Figure 2 shows timing change and real-time characteristic improvement with application of LOD. The time required for the drawing models A, B, and C 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 update of drawing contents in the next frame. Here, when B's importance is low, by switching simpler 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.

Many studies have been conducted regarding methods to simplify geometrical data[2]. Method to decide which detail level should be used at implementation greatly differs between variable and fixed frame-rate systems.

Detail levels are selected through individual processing data in variable frame-rate systems, while in fixed frame-rate systems, they are selected by a scheduler considering the total throughput and time restriction. Of course, fixed frame-rate systems have more of real-time characteristics.

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

On the other hand, recent improvements in hardware performance has led not only to the larger volume of graphical data, but also to the more complex 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 used in a program[6]. LOD processing can similarly be applied to such behavior describing, and using omitted behavior processing for objects with low importance on the screen when they are viewed in the distance can reduce the total throughput. However, the problem lies in that in most cases, a detail level switching 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

Scheduling systems for real-time processing have been widely studied. For example, RTC++[5] is an extension of C++ language with the following features:

- 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

This language supports explicit timing constraints, highly preemptable object, periodic task creation, and priority inheritance.

Such a framework is also applicable to 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 processing using lower quality is used. This is a very important characteristic in real-time processing, but general-purpose programming languages do not support automatic selection among alternative processing methods, thus programmers have to code such selection manually. Some software environments[1] can perform such selection, but they must use scene graph structures specific to the libraries and are not suitable when behavior processing also have to be selected according to the display selection.

4. Level of Detail Processing System

4.1 Language Design

We have explored mechanisms and specifications for such a language, which will be described below. We have also constructed experimental implementation for the language, and have evaluated usefulness of such language based on writing several example code.

For easy integration to current applications, the language was designed as minor expansions to C++, in which methods and statements have some extra functionality for LOD processing. Therefore, the grammar of the language mostly resembles that of C++. The implementation also outputs C++ code, which is compiled with ordinary C++ compilers.

4.1.1 LOD method definition

The following is class `A` declaration with method `f()`, which is exactly equivalent to C++ declaration.

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

However in our language, method `f()` may have multiple implementations, 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

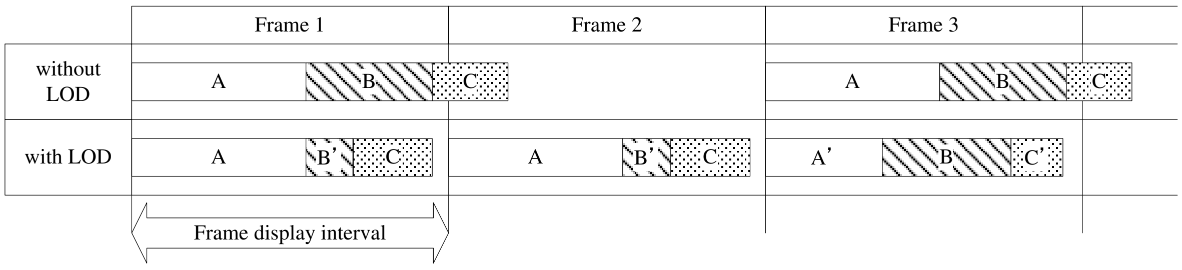


Fig. 2 Changing Real-Time Characteristics With or Without LOD Processing

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>
}
```

Now class A has a method `f()` with two LOD implementation. An appropriate implementation is chosen according to the context by calling simply method `f()`.

On the user side of this class object, LOD method is invoked from within a schedule block which specifies limit cost at the entry, in the following form:

```
schedule(limit){
 code to be executed
}
```

With such mechanism, we can easily select a process dynamically, just like as in the polymorphic method selection of object-oriented languages. For example, we can use methods of class A as follows:

```
/* -- Using objects -- */
{
  A a, b;
  schedule(200){
    a.f();
    b.f();
    /* the method with cost 100 is chosen
       for a and b */
  }
  schedule(100){
    a.f();
    b.f();
    /* the method with cost 50 is chosen
       for a and b */
  }
}
```

4.1.2 Object priority

In the above example, if the specified cost limit is 150, following two LOD combinations are possible:

- invoke `a.f{100}()` and `b.f{50}()`
- invoke `a.f{50}()` and `b.f{100}()`

In general, there will be many combinations that satisfy specified time limit. Like the shape LOD, we can have much better results with assigning higher cost to more important object.

There are many kind of importance criterion such as size of the corresponding object on the screen, or its speed of movement, or likewise. Exact priority assignment depends on application. We thought that single integer value for priority will be sufficient and reasonable. So the class with LOD method always has the instance variable `priority`, and the system will select among combinations of methods based on that value.

Here is code example that execute `a.f()`, `b.f()` and `c.g()` with total cost limit 200.

```
{
  A a, b;
  C c;

  a.priority = 5;
  b.priority = 3;
  c.priority = 2;

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

In this case, `a.f()` can choose the highest cost implementation because priorities are `a > b > c`.

Above is the basic outline of this language. Which implementation of LOD methods is actually used is determined dynamically. When we assign cost value proportional to processing time of each LOD method and choose appropriate limit values on schedule statements, scheduling for fixed frame-rate systems will be realized.

In the current design, no flow controls such as loop within a schedule block are allowed. However an nesting of schedule block is possible as in follows:

```
{
  schedule(200){
    a.f();
    b.f();
```

Table 3 State variables of moving vehicle

LOD	State variables
Low	Position
Mid	Position, Orientation, Velocity, Acceleration
High	Engine Rotational Speed, Tire Load, Transmission State, Steering Angle ...

```
    schedule(150){
      c.g();
      d.g();
    }
  }
}
```

In this case, a.f() and b.f() are executed with cost limit 50 because the cost to execute inside schedule block is guarantied at 150.

4.1.3 LOD Transition

In an actual application, different set of state variables are used for different LOD. For example, To describe moving vehicle needs various variables according to LOD (Table 3) .

When LOD changes, some variables for new LOD implementation should be initialized appropriately to maintain continuity. To avoid complicated branching in LOD method, the system records previous LOD selection and automatically invokes “transition methods” if necessary.

In the previous example, we can define a transition method as in the following:

```
/* -- LOD High -> Low Transition */
A::f{100->50}()
{
  <initialize variables of f{50}
    from variables of f{100}>
}
```

Similarly, reverse transition method can be defined:

```
/* -- LOD Low -> High Transition */
A::f{50->100}()
{
  <initialize variables of f{100}
    from variables of f{50}>
}
```

If transition method is defined for a transition, the method is automatically invoked before the main method for the new LOD. This mechanism resembles method combination feature seen in some of the object-oriented languages.

As a summary, in the proposed language, declarative description of multiple LOD code and transition code among them is possible, and runtime system automatically selects appropriate set of code and executes them.

4.2 LOD Selection

There are three factors in LOD selection:

(1) Method cost
statically specified in the source code.

(2) Cost limit
dynamically specified upon each entry of schedule blocks.

(3) Priority
associated to each object; can be changed dynamically through assignment to `priority` value.

The runtime system have to select appropriate LOD for each method invocation in schedule block. This is combinatorial optimization problem and it is difficult to have an optimal solution. So we took an approximate algorithm as follows:

(1) Proportionally distribute the cost limit to each invocation with their priority
(2) Select the highest cost implementation within the distributed limit
(3) Improve selection from higher priority object with surplus cost budget
(4) Repeat step 3 until there are no more higher cost selection within cost limit

Since LOD methods can be sorted by their costs at the compilation time, step 2 takes O(n) time to solve. Step 3 requires O(n log n) time in order to sort by priority.

5. Evaluation

5.1 Evaluation Criteria

We now shows the results of using our language in two example scenes. With these examples, we evaluate this language from the aspect of the easiness of description and the adequacy of LOD selection.

5.2 Sample 1: Eight Sphere

The first sample contains moving eight spheres in the grid plane. Each sphere is initially located at a random position and is moving along given direction at a constant speed. When they achieve an edge of the grid, they bound off the wall and keep moving in the grid. User can move the view point arbitrarily.

Figure 3 shows a screen shot A at some point of this application. There is a bar chart at the upper side and it shows cost counting of each sphere. In this situation, the cost limit is set to 100. There are three LOD of drawing sphere method and their costs are 30, 13, and 5.

The brown sphere is nearest to the eye point and it is drawn with the biggest shape in the screen. In this application, the priority used in LOD selection is set in proportion as the distance between the sphere and the eye point. Therefore the nearest brown sphere can have the highest LOD selection. The brown sphere is drawn with the highest LOD of cost 30 in screen A. The next red sphere is also drawn with the LOD of cost 30, and the next yellow sphere is drawn with the LOD of cost 13.

The upper row of Table 4 shows the LOD selection result in screen A. Total costs of inside schedule block invocations are 98 and it satisfy the limit of 100.

Figure 4 shows a screen shot B at another point of this application with different sphere positions and different eye point. The lower row of Table 4 shows the LOD selection result in screen B.

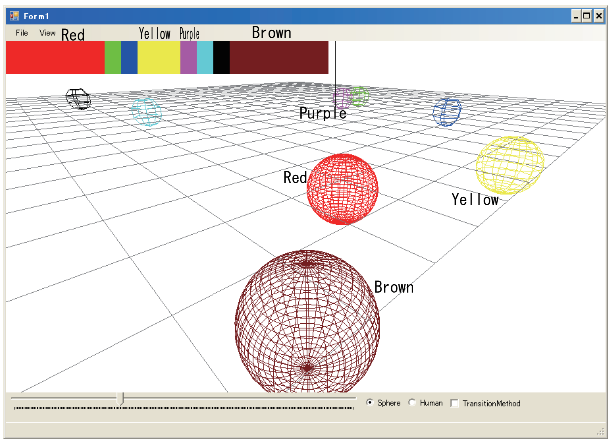


Fig. 3 Scene 1, Screen A

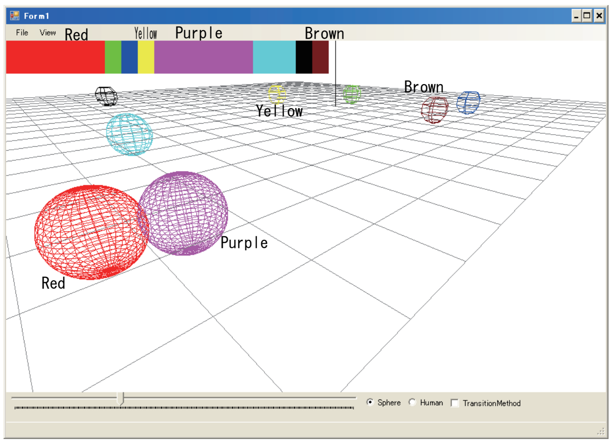


Fig. 4 Scene 1, Screen B

Like screen A, the cost total fits the given limit. In this case, the purple sphere is drawn with the highest LOD instead of the brown sphere because the brown sphere is far from the eye point and it doesn't require such a high detail at the moment. In this way, we have confirmed that the LOD selection should work depending on the priority. This example shows we can select LOD without complicated description in our language.

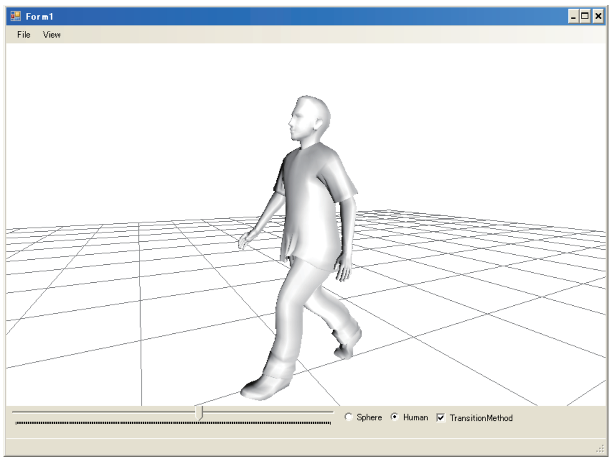


Fig. 5 Scene 2

5.3 Sample 2: Walking Person

Next we have examined how transition methods can be useful. In this scene, there is a human with walking animation (Figure 5)

The class which represent the human have two LOD selection (High and Low) to describe walking animation. Higher detail implementation `Walk{100}` consists a lot of parameter set such as foot position or knee angles or arm swing angle, etc. Lower detail implementation `Walk{50}` simply interpolate between two states – step left foot and step right foot – and it has only one parameter. These state variables are independent between `Walk{100}` and `Walk{50}`. In such case if the selected LOD is different from previous cycle, an noticeable flicker like a "flipping foot" can occur by the state discontinuity.

To address this problem, we have to initialize some state variables which will be used after a certain interval from other state variables which are used at previous cycle.

In our language, we can treat this problem simply writing transition methods as follows:

```
class Human
{
public:
  void Walk{50->100}()
    {
       /* initialize parameters for High
           from parameters for Low */
    }
  void Walk{100->50}()
    {
       /* initialize parameters for Low
           from parameters for High */
    }
}
```

Table 5 shows cyclic method invocation with these transition methods. You can see appropriate initialization methods are called when LOD transition will occur.

Table 4 LOD selection in screen A, B

Sphere	Red	Green	Blue	Yellow	Purple	Cyan	Black	Brown	Total	Limit
LOD in screen A	30	5	5	13	5	5	5	30	98	100
LOD in screen B	30	5	5	5	30	13	5	5	98	100

Table 5 method invocations with LOD transition

Cycle	LOD	Invoke Method
1	High	Walk{100}
2	High	Walk{100}

3	Low	Walk{100→50}
		Walk{50}

4	Low	Walk{50}
5	Low	Walk{50}

6	High	Walk{50→100}
		Walk{100}

7	High	Walk{100}

As a summary, we can have smooth LOD switching by just adding some initialization as a transition method instead of complicated conditional branches.

6. Conclusion

In this paper, we proposed programming language which support automatic LOD selection and transition. In contrast to 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) DavidLuebke etal. *Level of detail for 3D Graphics.* Morgan Kaufmann, San Francisco, 2003.
3) ClarkJ. H. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.
4) RHawkes. Update rates and fidelity in virtual environments. *Virtual Reality: Research, Development, and Application*, 1(2):99–108, 1995.
5) Yutaka Ishikawa, Hideyuki Tokuda, and CliffordW. Mercer. An object-oriented real-time programming language. *Computer*, 25(10):66–73, 1992.
6) 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.