A Software Architecture Based on the Idea of Instruction Set Architecture

Yang Cui

G28165607

Computer Architecture

Fall 2020

[Abstract] Graphics-Instruction-Model(GIM) is a modern software architecture that is heavily inspired by instruction set architecture. The core idea of GIM is to project machine level instruction into hyperplane and design an instruction set at application level. It offers the flexibility to break component logic into units and combine them dynamically during runtime.

Software architecture design has always been an issue. It is especially true for complicated GUI applications. This paper aims to purpose a Graphics-Instruction-Model(GIM) architecture for software design.

I. Motivation

Software design was not an important thing when computer first came out. It's all about algorithm design. Even nowadays batch program doesn't need too much design efforts. However, when the concept of graphics user interface was introduced, software design became a problem. As user demands grow more and more complicated, so is the software. A good design or framework can save a lot of efforts in future updates.

There are already some existing frameworks for GUI application. One famous family includes model–view–controller(MVC) and its modified version model–view–presenter(MVP). Other approaches include event-driven and component-based architecture.

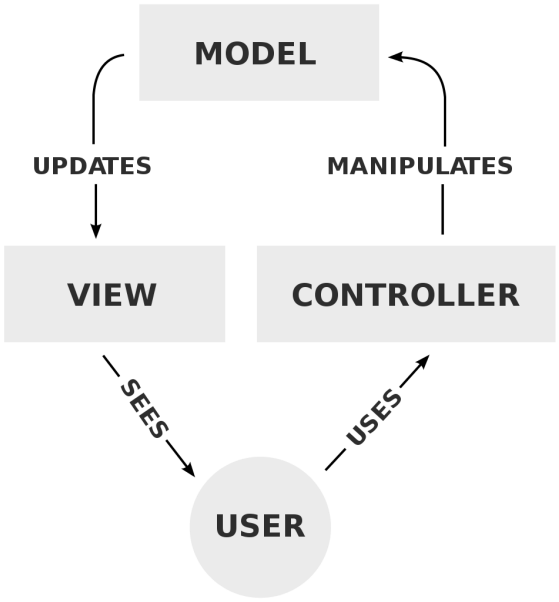

Figure 1 Diagram of MVC[1]

[1] RegisFrey, released on WikiMedia under public domain license.

MVC splits the application into three major components.
View is what rendered on screen. For example a chart or an image. In a broader sense, it is the graphics user interface that users see.
Model is the data, logic and rules of the application. It is separated from views. In other words, you can have different views for the same model.
Controller is the component that connects the user and model. It receives the user input and tells the model to act accordingly.

As you can see, the biggest advantage of MVC is the separate of duty. Each component can change independently as long as they act according to protocols. However, there are some disadvantages of MVC. The biggest one can be summarized as the following: MVC is a traditional architectural pattern. It can't handle modern GUI application requirements.
What is the requirements of modern GUI application? In general, a modern GUI application has the following characteristics.
1) Interactive
2) Multimedia
3) Multithread
4) Continuous IO

For interactive part, you may ask that isn't traditional GUI application interactive too? By interactive I'm not saying it's just click a button and trigger an event. User interaction can be quite complicated. For example in a 3D simulator, the viewport will change according to the cursor. There are also UI buttons to let users operate the application. There're also keyboard inputs and audio input. If you want to tie all these things together and make it scalable and flexible for future updates, MVC will not meet the goal.
For multimedia part, modern GUI application is expected to handle 2D, 3D, audio and video streams. Possibly animations too. In the MVC architecture, these all all handled inside the model, which can make the model grow too fast. It is also a typical pitfall for MVC architecture.

Almost all modern GUI applications run in a multithread environment. Threads are powerful but at the same time dangerous. It is best to take it into consideration when designing architecture. Obviously threading is not like a function that can be easily added or removed. MVC is purposed at a time when multithread was not prevalent. Thus is can't address this problem nicely.
IO here could be anything from a socket to a file. And it is important to notice that the IO is continuous, which implies that the IO is not predefined as well. An online video player receives audio stream continuously, but this is actually not too hard to build. The IO is predefined. What I mean here by "not predefined" is similar to a web browser. A web browser will get the web pages and render them on screen. Different web pages have different functionalities but the web browser can handle them. This process is more than just rendering. There are also interpretation going on. Clearly MVC can't do this.

To meet these requirements in modern GUI application, I am purposing GIM architecture in this paper. GIM can be understood as something that combines both the idea of MVC and ISA.

II. Designing Instruction Set Architecture

GIM is heavily inspired by ISA. In fact the computer itself is the best example that can handle unpredefined functionalities. At first this seems contradictory. As we know each instruction is precisely defined. If not computer can't execute anything. However, computer can also run whatever applications it sees. How can a machine that only executes precisely defined instructions is able to run any applications?

A. Inspiration from ISA

Although there are different instruction design choices, an abstract instruction can be split into three portions: opcode, operand and address.

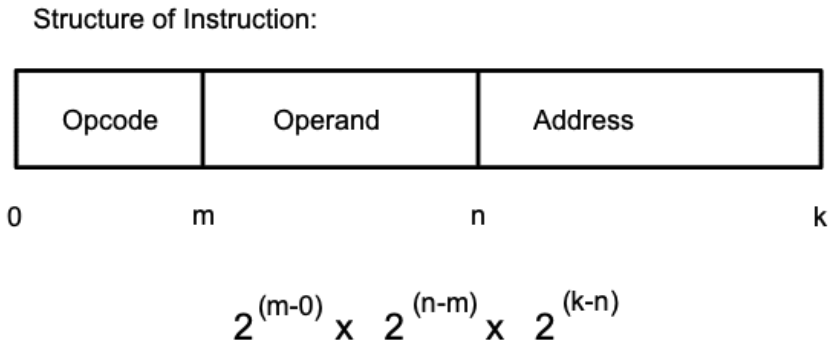

Figure 2 Sructure of Instruction

Each portion has a different length of bits. If we multiple them together we can calculate the maximum number of combination for a given instruction length. In theory if we just extend the length of instruction we can express anything. But longer instruction means more processing time and extra cost on the circuit. So generally we keep the instruction to 64 bits.

This, however, doesn't answer the problem raised before. If we think from the application perspective, applications can achieve anything we can think of. This means the functionality space is infinite. How can a finite instruction expression space meet the requirements of infinite space?

In fact the power is not inside the combination of one instruction, but the combination of many instructions. We can stack as many instructions as we want into one file, which is the program, and execute them. In this way we have an infinite number of instructions to meet the infinite requirement space. Since one instruction can potentially express all available operations the computer can provide, our program can do all things that is possible with a computer.

B. Project Instruction into Hyperplane

As one famous idiom goes, "we can study the bird to learn how to fly, but airplanes doesn't need to be built like a bird". The ISA allows us to operate the machine in a lower level, like load

contents from memory to register, or add two numbers up. However, if we just replicate these instructions and use them in application level, the programing work would be too painful. There are several semitic planes in this space. What we should do is to project the instruction into a hyperplane to get the instructions we use at application level.

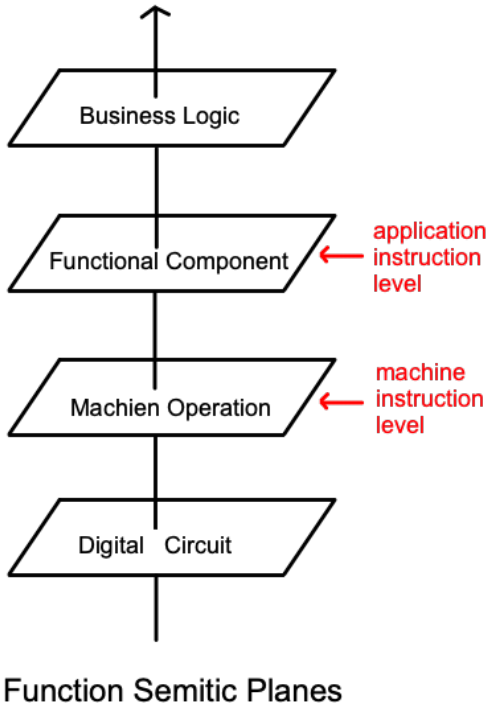

Figure 3 Function Semitic Planes

The question becomes how do we project from machine instruction to application instruction? First, we keep the overall structure of instruction. That is to say we still split one instruction into several portions. As to exactly how many portions are needed really depends on the business logic you are implementing. Usually this needs to be determined as design phase. But as we will see later, the instructions at application level are pretty flexible and mutable. You can always change the design later. This is a sharp contrast to machine level instruction.

Next we should specify a format for the instruction. This format is not the structure of the instruction. Structure is more about conceptual view of the instruction, while format specifies how to organize the instruction in a meaningful manner. For example you can use json or html as the format. Usually this is good choice because the Web world has already proved the feasibility of this approach. You can also come up with custom format as long as you keep it consistent. Another thing worth mentioning is that we do not need to use bit to specify the information. We can just use text to convey the meaning of instruction. This is a more reasonable approach in application level. This provides good readability and enables us to debug with ease.
Then we need to write the instruction set as a design document. The document needs to specify what instructions are needed and the corresponding descriptors. Instructions can be easily added so there is no need to try to exhaust all instructions.
Finally we implement these instructions in an OOP manner. The typical way is to design an interface/abstract class Instruction. Then use derived class to implement the logic of each instruction. We can use factory pattern to instantiate derived instructions.

The biggest benefit for doing this is that we have separated logic with application. What does it mean? Think about machine instruction. Instructions just carry some information. It is the ALU that performs the actual logic, like adding or multiplying. In this regard the derived instruction class is the ALU. It wraps the component logic in a single place and can be called on demand. If we achieved this, the role of application also changes. In the past the application needs to consider the logic of both component and business. Now it only needs to combine instructions to achieve business logic.

This change of behavior will offer many advantages. It is easy to maintain and extend, as we have placed one component logic in one class. It is easy to reuse the instructions to build many applications, since each application now only combines instructions. It serves as a base to meet all modern GUI application requirements. I will show this point when we discuss the overall framework in the next section.

C. Example of 3D Designer Application

Now I will demonstrate an example to show the application ISA in action.
Suppose you are developing a 3D designer application that lets users place communication towers in space. This software can be used by communication companies to calculate the cost and covered area when planning to setup new towers.

Let's just focus on one function of the software: placing a tower in 3D space.
To achieve this function we need several instructions.

	Op	Obj	Desc
Creating tower	41	Stirng	
Moving tower	42	Stirng	x-int, y-int, z-int
Rotating tower	43	Stirng	x-int, y-int, z-int
Modeling tower	44	Stirng	path-string

Table 1 Examples of Instruction Set

The instruction is split into three parts. Op is an integer that denotes which instruction it is. Obj is the object name in string. Desc will specify the parameters of this instruction.

Suppose we use json as the format. Then to place a tower we need first to create one, move it to desired place, set tis rotation and set the model to be rendered. As a result we will need these instructions.

{"Desc":null,"Obj":"TowerA","Op":41}
{“Desc":{"x":1000,"y":700,"z":50},"Obj":"TowerA","Op":42}
{“Desc":{"x":60,"y":180,"z":0},"Obj":"TowerA","Op":43}
{"Desc":{"path":"Data/model/tower_com_lv1.mdl"},"Obj":"TowerA","Op":44}

These four instructions will achieve the goal of placing a tower in 3D space.
Notice the big thing here. These four instructions can be contents from a file, input from user or message from socket. Where it comes doesn't matter. The application will just load these instructions and execute. Then users will see a tower on screen.

However, the instruction set alone won't help us meet the requirements of modern GUI application. How do we exactly load and execute instructions? We need a framework to do this. GIM is built for this.

III. Designing Software Architecture

I already covered the instruction part in GIM. In this section I will explain the graphics and model part, and how to put them together.

A. Overall Structure

The whole GIM framework can be illustrated by the following diagram.

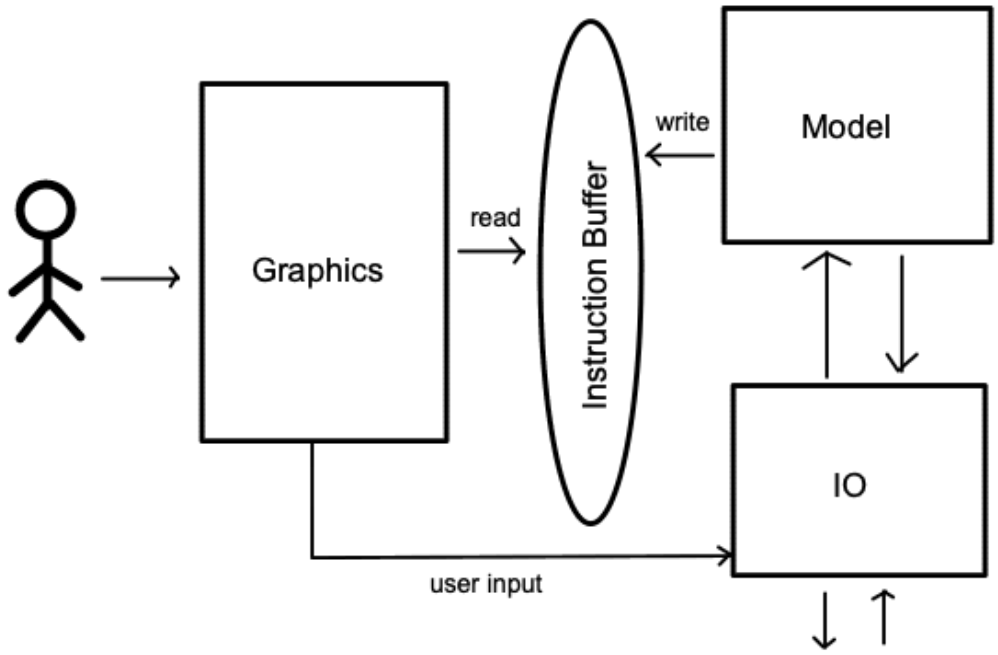

Figure 4 Framework of GIM

Graphics is the user interface that user can see and operate. Model is the data and logic of the application. IO is an abstract portal that does IO related functions. Instruction buffer holds instructions that will be executed.
At first glance the instruction buffer part seems strange. Only the model will write instruction into the buffer. Only the graphics will read from the buffer. This suggests that both model and graphics are actors. In fact you can implement them as different threads. Or a better idea is to let the application do a main loop. For each loop step you let graphics read from buffer and model write into buffer. No matter which approach, it is better to make the buffer mutual exclusive. This will make multithread environment safe.

The question is where is instruction executed? If only the graphics can read from buffer, the instructions are executed inside graphics. This is exactly the reason why I am not using the word "view", but rather choose "graphics" instead. This graphics represents the GUI library you use. Its job includes rendering, receiving user inputs and possibly the main loop. For the rendering work it is the same with other framework. For the receiving user input work, it will deliver this input into IO and let IO pass it into model. This is different from the popular callback methods used in GUI library. For the main loop part, each step graphics reads all instructions in the buffer, instantiate derived class and call the API to execute the instructions. The execution of instructions can generate more instructions. That's how the framework keeps going.

B. Sequential Diagram of Instruction Execution

The most important part of GIM is the read and execute instruction part. It deserves a diagram. Suppose the main loop will call Update() during each step, the following sequential diagram will demonstrate the complete cycle of one instruction.

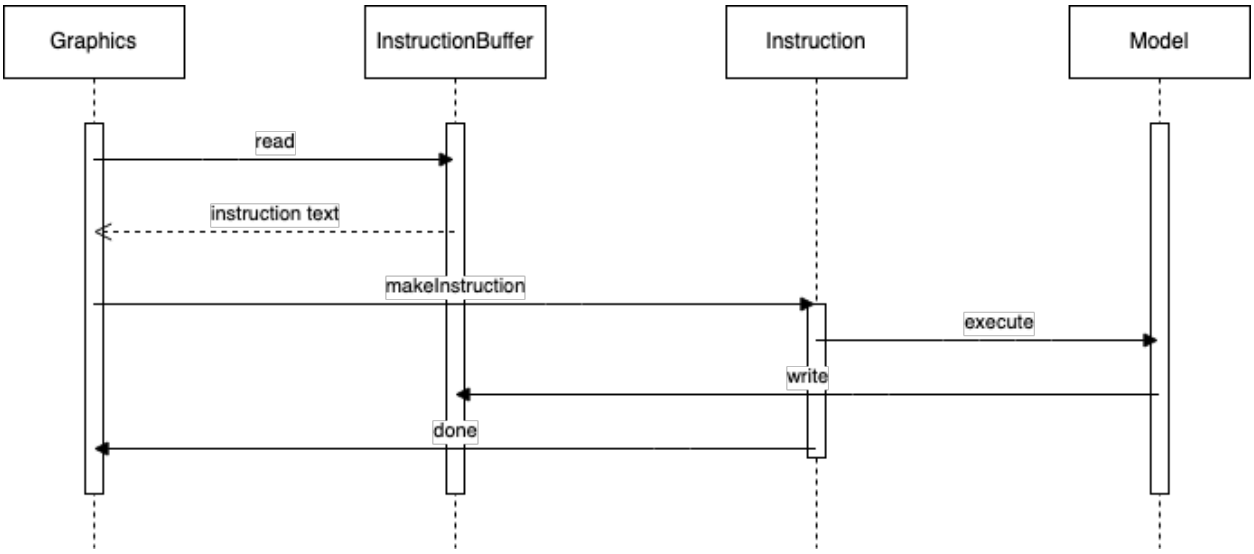

Figure 5 Sequential Diagram of Instruction Execution

Inside the main loop, we first read the instruction from buffer. Next using the information we instantiate the derived class and execute. During execution more instructions will be written into buffer. These newly written instructions will be executed in the next update.
This process is similar to instruction execution at machine level, where we have fetch, decode, execute, write back and update PC.

It is worth pointing out that during the execution of one instruction, both model and graphics can be modified. The aim of GIM is not separation of data and view. Its goal is to group component logic into one unit and make it flexible.

C. Discussion on Model

Model includes data and logic of the application. But if we have already put logic into instructions, what is the logic inside model?

The logic inside model is strictly business logic. Take the 3D tower designer for example. Traditional architecture don't distinguish component logic and business logic. Drawing a tower in 3D space is the same as calculating the cost of a tower. But in GIM, drawing a tower is component logic and has nothing to do with business logic. Calculating the cost of a tower is the real business logic. The logic inside model is the latter.
The advantage for this separation is obvious. Drawing a tower usually won't change once the GUI library is determined. The cost of a tower is prone to change. This separation makes future maintenance cost as little as possible.

The data inside model can be structured to business need, and should be organized in a meaningful way. This is especially true when the instruction involves change the status of an object by some degree. For example if you want to move an existing tower up by 50 pixels along the z-axis, it is easy to write this in instruction set and actually hard to implement. Why? Because instruction itself has no knowledge of existing status. Besides you can't get this information from graphics, since the job of graphics doesn't include storing the data. As a result you need to rely on model for the data. In fact the model should correspond to the graphics all the time. If they don't match, something is wrong.

IV. Evaluation

Can GIM meet all the requirements of modern GUI application?

A. Four Criteria

The graphics part receives user input and pass it into IO and further down to the model. During this process more instructions can be written into buffer. These instructions will be executed in the next update and change graphics and model. In this regard GIM is interactive. But there is

more than this traditional interaction. GIM can dynamically interpret, build and add functions to UI layout. In fact it provides a more flexible interactive environment than traditional methods.
GIM can certainly play multimedia resources. No matter it is image, animation, 3D model, video, audio or something else, we can always add more instructions to play that type of media resource. There may be a concern on how different media resources are integrated into the user interface. This question depends on the GUI library you use. Some libraries provide full support and that is the ease case. You just need to wrap these functions inside the instructions and done. If not, say 3D viewport, you may need to come up with your own way to do it. Instructions don't provide functions. They just break functions into unit and let you combine these functions in a flexible way.
GIM supports multithread environment perfectly. The core connection point is the instruction buffer. As long as the buffer is mutex, you can have as many threads as you want, each writing to the buffer continually. Graphics doesn't care who writes into buffer. It only reads the buffer and execute the instructions. You can also make IO a different thread that keeps listening to socket, receives instructions and write into buffer. But you still need to consider the problem of synchronization if the order of instructions matters.
GIM can receive continuous and unpredefined IO, and interpret them dynamically. In fact if we provide all necessary instructions, it can even reach the flexibility of a web browser.
In conclusion GIM can meet all the requirements of modern GUI application.

B. Scripting

If the instruction set is comprehensive enough, what you finally get is not an application but an engine. This should not be a surprise. Given a bunch of instructions, be them from network or file, the engine interprets them and execute accordingly. The users may use the same engine like an editor or play video games. These can all be achieved with the correct instructions.
This provides the possibility of scripting outside the engine. Scripts are just plain text files, but when interpreted by a script engine it can run code logic. Now if we generate instructions from scripts and let the engine read these instructions, we literally moved business logic out of the application itself. In other words, we can use the combination of script and engine to achieve

anything the instruction set can provide. Now we can change business logic without the need to recompile the application. This can come in handy under certain circumstances.

If we go one more step and make the scripts into byte codes, the engine can potentially become a virtual machine. Think of the example of JVM. You write some classes, you compile the classes into byte codes and then pack them into a jar file. Others can run this jar file as an application. GIM can support this process without problem. You write some scripts, compile them into byte codes and pack them together. Now you run this package. It generates instructions and the engine reads, interprets and execute them. The process is almost identical to how JVM works.

C. Drawbacks

There are something to consider before choose the GIM architecture.

The first thing is that does the application really need this much flexibility? For simple applications, like a plain text editor, using traditional architecture can be a good choice. GIM can be harder to design and implement. Is it worth the efforts to gain the flexibility? If you want to build an application that is closer to an interpreter, GIM will be a good idea.

The second thing is efficiency. Flexibility always comes with a cost. During each loop step, there could be a large amount of instructions that needed to be executed. Suppose each loop takes about 0.016 seconds, then all instructions, no matter how many there are, should all be finished within 0.016 seconds. If not the user will experience fps drop. You may limit how many instructions can be executed in one loop. This, however, introduces overhead. Besides, the hardwares differ from machine to machine, it is hard to estimate the maximum number of instructions per loop.

If you use multithread, some parts need to have mutex. These operations can cost much more computation resource than non-mutex operations. These matters are also something to think about.

Third, the security vulnerabilities in GIM can be enlarged. This is due to the interpreter nature of this architecture. Let’s take removing a file for example. In traditional architecture, usually the user click a button to remove a file. Since the context is determined, you can always do sanity check in the codes. However, in GIM if you design one instruction that can remove a file, sanity

check may not be possible because instructions have no idea about context. It just does what it is written there. You either need to add parameters into the instruction to inform the engine about the context, or you redesign instruction set so that removing a file is not a standalone instruction. In either way there is more to consider.
This especially holds true if you decide to go the scripting way. Script can do complicated code logic, but the engine knows nothing about script logic. It just reads instructions generated from scripts and execute. As a result you need to be very careful about the security problems introduced by flexibility.

In conclusion GIM is a modern software architecture and very suited for interpreter-like engines, like a web browser. It offers the flexibility to break component logic into units and combine them dynamically during runtime. It is also possible to move business logic out of application via scripting.

References

Yonglei Tao, "Component- vs. application-level MVC architecture," 32nd Annual Frontiers in Education, Boston, MA, USA, 2002, pp. T2G-T2G, doi: 10.1109/FIE.2002.1157950.

M. Gu and K. Tang, "Comparative analysis of WebForms MVC and MVP architecture," 2010 The 2nd Conference on Environmental Science and Information Application Technology, Wuhan, 2010, pp. 391-394, doi: 10.1109/ESIAT.2010.5567323.

Yul Chu, "A simple project for teaching instruction set architecture," Fifth IEEE International Conference on Advanced Learning Technologies (ICALT'05), Kaohsiung, 2005, pp. 69-71, doi: 10.1109/ICALT.2005.23.

I. Barbieri, M. Bariani, A. Cabitto and M. Raggio, "Multimedia-application-driven instruction set architecture simulation," Proceedings. IEEE International Conference on Multimedia and Expo, Lausanne, Switzerland, 2002, pp. 169-172 vol.2, doi: 10.1109/ICME.2002.1035539.