Driver Development

**Note:**

**Table of Contents**

[1. Overview 2](#_Toc443242327)

[2. Server-side development 3](#_Toc443242328)

[2.1 Render driver 3](#_Toc443242329)

[2.2 Part driver 4](#_Toc443242330)

[2.3 Component driver 8](#_Toc443242331)

[2.4 Instance driver 9](#_Toc443242332)

[3. Client-side development 10](#_Toc443242333)

[3.1 Function parameters 0](#_Toc443242334)

[3.2 Render Variable 1](#_Toc443242335)

## Overview

Development comprises of three parts:

1. Server-side driver development: developing programs that run on web server, the development language is Java. This development includes development of four Java classes: render driver, part driver, component driver, instance driver.
2. Client-side JavaScript program development: developing program that runs in web browser, the development language is JavaScript. It is in this development that we use WebGL to draw 3D objects. This development includes development of two JavaScript function: one for drawing 3D objects, the other for decoding data from server, usually decoding function create frame data from face data.
3. GPU-side Shader development: developing shader programs that run on GPU, its development language is GLSL. At present, this development need implement two type of shader: vertex shader and fragment shader. In the future, we probably need implement other shaders, such as geometry shader, tessellation-control shader, tessellation-evolution shader, computation shader, etc.

There is a development example: the example driver. It is very simple. You can learn how to do development from example driver by reading its codes.

## Server-side development

This development need implement four drivers in Java.

1. Render driver: render correspond programs.
2. Part driver: Render driver creates part driver. Part correspond geometry, shape
3. Component driver:
4. Instance driver:

### Render driver

Example:

package driver\_lession\_2018\_00\_start\_driver;

import kernel\_driver.part\_driver;

import kernel\_driver.render\_driver;

import kernel\_engine.system\_parameter;

import kernel\_file\_manager.file\_reader;

import kernel\_network.client\_request\_response;

import kernel\_part.part;

public class extended\_render\_driver extends render\_driver

{

public void annihilate()

{

super.annihilate();

}

public extended\_render\_driver()

{

super();

}

public part\_driver create\_part\_driver(file\_reader part\_fr,part p,

system\_parameter system\_par,client\_request\_response request\_response)

{

String meterial\_file\_name=p.directory\_name+p.material\_file\_name;

return new extended\_part\_driver(meterial\_file\_name);

}

public void get\_parameter(file\_reader render\_fr,

system\_parameter system\_par,client\_request\_response request\_response)

{

}

}

Only three methods need be implemented:

1. annihilate: Java Virtual Machine implements automatically garbage collection, any java class has no destructor method, hence we provides annihilate method. When an java object will not be used, engine will call its annihilate method to release its resource. In this way, engine avoids memory leak.
2. constructor: it is inherited from base class render\_driver. This method needs no parameter. When engine processes Shader Configuration File(see configuration document), it uses this class name to identifies render driver.
3. create\_part\_driver: each part has a part diver to manage it. When engine processes Part List File(see configuration document), it create all parts as well as their drivers through this method. The method has four parameters, the first is a file\_reader object of Part List File(see configuration document), the second is the part object, the third is system parameter object of engine, the last is client request response, it is employed for getting parameter from client-side browser.
4. get\_parameter: if you need some user-defined parameters when create part diver, you can define them here. When engine processes Render List File(see configuration document), after it has read System Render Parameter File and Part List File, it call get\_parameter() to load user-defined parameters here. The method has three parameters, the first is a file\_reader object of Render List File, the second is system parameter object of engine, the third is client request response, which is employed for getting parameter from client-side browser.

### Part driver

Example:

package driver\_lession\_2018\_00\_start\_driver;

import kernel\_camera.camera\_result;

import kernel\_component.component;

import kernel\_component.component\_link\_list;

import kernel\_driver.component\_driver;

import kernel\_driver.part\_driver;

import kernel\_engine.client\_information;

import kernel\_engine.engine\_kernel;

import kernel\_engine.system\_parameter;

import kernel\_file\_manager.file\_reader;

import kernel\_file\_manager.file\_writer;

import kernel\_network.client\_request\_response;

import kernel\_part.part;

import kernel\_part.part\_container\_for\_part\_search;

import kernel\_part.part\_rude;

import kernel\_transformation.box;

import kernel\_transformation.point;

public class extended\_part\_driver extends part\_driver

{

private String material\_file\_name;

public void annihilate(part p,engine\_kernel ek)

{

super.annihilate(p,ek);

}

public void initialize(part p,engine\_kernel ek,client\_information ci)

{

super.initialize(p,ek,ci);

}

public extended\_part\_driver(String my\_material\_file\_name)

{

super();

material\_file\_name=new String(my\_material\_file\_name);

}

public part\_driver clone(part p,system\_parameter system\_par,client\_request\_response request\_response)

{

return new extended\_part\_driver(material\_file\_name);

}

public int caculate\_material\_id(part p,String type\_str,

String material\_x,String material\_y,String material\_z,String material\_w)

{

return 0;

}

public part\_rude create\_mesh\_and\_material(part p,file\_writer buffer\_object\_file\_writer,

part\_container\_for\_part\_search pcps,system\_parameter system\_par)

{

file\_writer.file\_copy\_with\_brother(

p.directory\_name+p.mesh\_file\_name,buffer\_object\_file\_writer.directory\_name);

return super.create\_mesh\_and\_material(p,buffer\_object\_file\_writer,pcps,system\_par);

}

public component\_driver create\_component\_driver(file\_reader fr,

part my\_component\_part,engine\_kernel ek,client\_request\_response request\_response)

{

return new extended\_component\_driver(my\_component\_part);

}

public box caculate\_part\_box(

part p,component comp,int driver\_id,int body\_id,int face\_id,int loop\_id,int edge\_id,int point\_id,

point p0,point p1)

{

return super.caculate\_part\_box(p,comp,driver\_id,body\_id,face\_id,loop\_id,edge\_id,point\_id,p0,p1);

}

public void create\_component\_render\_parameter(part p,

component\_link\_list delete\_cll, component\_link\_list refresh\_cll,

component\_link\_list keep\_cll, component\_link\_list append\_cll,

engine\_kernel ek, client\_information ci, camera\_result cam\_result)

{

}

public String response\_event(part p,engine\_kernel ek,client\_information ci)

{

return super.response\_event(p,ek,ci);

}

}

Part driver is the most complex driver. Nine methods need be implemented:

1. Annihilate: Java Virtual Machine implements automatically garbage collection, any java class has no destructor method. When any java object will not be used, engine will call this annihilate method to release its resource. In this way, engine avoids memory leak.
2. Initialize: After a scene has be created, engine call this method of all parts to do initialization. This method can response some data to client-browser. The response data will stored in part\_initialize\_data variable of javascript function.
3. Constructor: it is inherited from base class part\_driver. Base class (part\_driver) need no parameter. If you need other parameters, you get them from render driver in part driver constructor. In this example, we need a material\_file\_name, render driver get it from part object and pass it to part driver in constructor method.
4. Clone: clone a new part diver object from an existing one. Different scene may create component object from same part object. It probably causes some conflicting problems when different scenes access same part driver synchronously. Therefore when a new scene is created, new part drivers that are private to the scene is created by this clone method. If there is no conflicting problem, the cone method can just return this.

Clone method has three parameters. The first is the part object, the second is system parameter object of engine, the third is client request response, it is employed for getting parameter from client-side browser.

1. create\_mesh\_and\_material\_file: part object has a part\_mesh field; it is part’s geometry information, which is configured in Part Mesh File (see configuration document). The file name is p.directory\_name+p.mesh\_file\_name, where the p is part object. If its file format is engine standard format, or obj, or stl, this method need only inherit from its super method. If it is not, you can create an part\_rude object in this method by yourself.

This method has four parameters. The first parameter is its part object; the second parameter is file\_writer object of buffer\_object header file. If a part driver need create some files that is private to its part, it can create the private files in directory of the file\_writer. Client-side JavaScript program can download the private files by calling response\_event method of a part driver. The third parameter is a part\_container\_for\_part\_search object, from which you can find other part. If you need some data of other part in this method, you can find them by search\_part method of the part\_container\_for\_part\_search object. The fifth parameter is system parameter object.

Usually a part driver writes some material information into the file\_writer. In this way, part driver transfers material information to Javascript program running on client-side browser.

The return value of this method is a part\_rude object.

1. create\_component\_driver: when a scene is created, the engine creates component objects by calling this method of part drivers. This method has four parameters. The first parameter is a file\_reader object whose file is Component Assembly File(see configuration document). You can define some component private data in Component Assembly File and read them in create\_component\_driver method here. In this way, engine passes private data in constructors of component drivers. The second parameter is part object of this part driver. The third parameter is engine\_kernel object of the scene, the forth parameter is client\_request\_response request\_response, from which you can get parameter from client-side users.
2. caculate\_part\_box: create part’s bounding box in local coordinate of the part. This method has ten parameters. The first is part object of the part driver. The second is a component object. A component may have none or several component drivers, the third parameter is component driver ID which identifies it. The component object may be null, component driver ID may be less than zero, all these means that there is no component relate to the part. The forth to eighth parameters are body ID, face ID, loop ID, edge ID, and point ID. They identify interior information in a part. The ninth and tenth parameters are two points that identify a view line. The view line may cross the part with two or more cross point. In this case, the caculate\_part\_box method return a box of the cross point that is near to viewer.
3. create\_component\_render\_parameter: Client-side JavaScript program in browser caches component parameters to reduce parameter data transmission. If component parameters are modified on web server, engine will update cached parameters in browser. Some components with same part may share some common parameters. If each component driver updates its shared common parameter separately, same updating data may be transmitted several times. If these shared common parameters are transmitted to JavaScript program in create\_component\_render\_parameter method of part driver, same updating data will be transmitted only once, therefore updating data transmission will be reduced.

The create\_component\_render\_parameter method has eight parameters. The first parameter is part object. The second to fifth parameters are component link list. There are deleted component link list, refreshed component link list, kept component link list, appended component link list. The sixth parameter is scene data, its name is ek, and its data structure is engine\_kernel. The seventh parameter is client information, its name is ci, and its data structure is client\_information. The last parameter is camera related data, its name is cam\_result, and its data structure is camera\_result.

1. response\_event: part driver can response calls from client-side JavaScript program by response\_event method. Client-side JavaScript program call this method by Ajax, its function is

render.call\_server\_part (

render\_id,part\_id,

part\_parameter,

response\_function,

error\_function);

### Component driver

Example:

public void annihilate(component comp,int driver\_id,engine\_kernel ek)

{

super.annihilate(comp,driver\_id,ek);

}

public extended\_component\_driver(part my\_component\_part)

{

super(my\_component\_part);

}

public void initialize(component comp,int driver\_id,

engine\_kernel ek,client\_request\_response request\_response)

{

// String component\_directory\_name=comp.component\_directory\_name;

// String scene\_directork\_name=ek.scene\_directory\_name;

// String parameter\_directory\_name=ek.scene\_par.directory\_name;

return;

}

public instance\_driver create\_instance\_driver(component comp,int driver\_id,

engine\_kernel ek,client\_request\_response request\_response)

{

return new extended\_instance\_driver(comp,driver\_id);

}

Component driver is very simple. Only two methods need be implemented:

1. Annihilate: Java Virtual Machine implements automatically garbage collection, any java class has no destructor method. When any java object will not be used, engine will call this annihilate method to release its resource. In this way, engine avoids memory leak.
2. Initialize: After a scene has be created, engine call this method of all components to do initialization. This method can response some data to client-browser. The response data will stored in component\_initialize\_data variable of javascript function. The component\_initialize\_data variable is an array indexed by buffer\_id. Every element is initialization data of a component.
3. Constructor: it is inherited from base class component\_driver. The component\_driver’s constructor need only one parameter, which is part object of the component driver. Engine calls create\_component\_driver method of part driver to create a component driver.

To support LOD(level of detail), a component may has several component driver. According LOD value, engine selects a suitable component driver for scene rendering.

1. create\_instance\_driver: This method creates an instance driver. A scene may be drawn by several browsers on different computers. An instance is an object corresponding to a component on one browser. The create\_instance\_driver method needs four parameters. The first is component object, the second is component driver ID, the third is scene data, its name is ek, and its data structure is engine\_kernel. The forth is client\_request\_response request\_response, from which you can get parameter from client-side users.

### Instance driver

Example:

package driver\_lession\_2018\_00\_start\_driver;

import kernel\_camera.camera\_result;

import kernel\_component.component;

import kernel\_driver.instance\_driver;

import kernel\_engine.client\_information;

import kernel\_engine.engine\_kernel;

public class extended\_instance\_driver extends instance\_driver

{

public void annihilate(engine\_kernel ek,client\_information ci)

{

super.annihilate(ek,ci);

}

public extended\_instance\_driver(component my\_comp,int my\_driver\_id)

{

super(my\_comp,my\_driver\_id);

}

public void initialize(engine\_kernel ek,client\_information ci)

{

super.initialize(ek,ci);

}

public boolean check(engine\_kernel ek,client\_information ci,camera\_result cr)

{

return false;

}

public void create\_render\_parameter(int buffer\_id,engine\_kernel ek,client\_information ci,camera\_result cr)

{

ci.request\_response.print(buffer\_id);

}

public void create\_component\_parameter(engine\_kernel ek,client\_information ci)

{

ci.request\_response.print(comp.component\_id);

}

public String response\_event(engine\_kernel ek,client\_information ci)

{

return null;

}

}

Instance driver has five methods:

1. Annihilate: Java Virtual Machine implements automatically garbage collection, any java class has no destructor method. When any java object will not be used, engine will call this annihilate method to release its resource. In this way, engine avoids memory leak.
2. Initialize: After a scene has be created, engine call this method of all instances to do initialization. This method can response some data to client-browser. The response data will stored in instance\_initialize\_data variable of javascript function. The instance\_initialize\_data variable is an array indexed by buffer\_id. Every element is initialization data of a component instance.
3. Constructor: constructor is inherited from base class instance\_driver. Base class (instance\_driver) needs two parameters. The first parameter is its corresponding component object. The second parameter is component driver ID of the component object.
4. Check: When engine collects instance for rendering, it traverse all instances in scene tree. If an instance driver wishes to render the instance, it can set variable instance\_check.abandon\_display\_flag false. Otherwise set it true.

Check method has three parameters. The first is scene data, its name is ek, its data type is engine\_kernel ek; The second parameter is client-related data, is name is ci, its data type is client\_information; The last parameter is camera-related data, its name is cr, its data type is camera\_result.

1. create\_buffer\_parameter:
2. create\_render\_parameter:

To reduce transmission of rendered data, client-side and server-side program cooperate together to cache rendering parameters of all components. If a component‘s rendering parameter has been modified at server-side, it should transmit the modification to client-side program.

To draw a scene, engine needs multi-pass rendering. Some render for texture, some render for selection, some render for shadow, etc. Different rendering pass may need different parameters.

The create\_buffer\_parameter method of an instance object does the modification transmission of multi-pass shared parameters, while the create\_render\_parameter method does that of single pass-specific parameter for a rendering-pass.

Both methods have two or three parameters. The first parameter is scene data, its name is ek, and its data structure is engine\_kernel. The second parameter is client information, its name is ci, and its data structure is client\_information.The last parameter is camera related data, its name is cr, and its data structure is camera\_result.

1. response\_event: instance driver can response calls from client-side JavaScript program by response\_event method. Client-side JavaScript program call this method by Ajax, its function is

render.call\_server\_component(

component\_name\_or\_id,driver\_id,

component\_parameter,

response\_function,

error\_function);

## Client-side development

### How scene is rendered by WebGL

A scene is rendered by WebGL as following steps.

1. download shader source (vertex shader and fragment shader) from webserver, compile and link them into programs.
2. create WebGL buffer objects, download primitives data from server and store them in buffer objects.
3. Identify shader program and buffer objects for rendering.
4. Set values for uniform variables in shader program
5. WebGL is an OpenGL state machine. Set necessary state values for WebGL.
6. Issue drawing orders.

Step 1 and step 2 only execute at engine starting and is implemented by on client-side engine.

Step 3 to step 6 is execute for every scene rendering. Client-side development need only implement one function for Step 3 to step 6 in JavaScript . The drawing function is called from client-side engine for every part, drawing all the components of that part. The function’s format is like this.

### Client-side JavaScript Example function and variables

function ( method\_id, parameter\_channel\_id,

render\_id, part\_id, render\_buffer\_id,

component\_data\_array, component\_part\_data, project\_matrix,

do\_render\_number, render)

{

var gl =render.gl;

var shader\_program =render.render\_program.render\_program[render\_id].shader\_program;

var part\_buffer\_object =render.buffer\_object.buffer\_object[render\_id][part\_id];

var component\_location =render.component\_location\_data;

var data\_buffer =render.data\_buffer[render\_id][part\_id];

var pickup =render.pickup;

var target =render.target;

var part\_information =render.part\_information[render\_id][part\_id].information;

var part\_material =render.part\_information[render\_id][part\_id].material;

var part\_property =render.part\_information[render\_id][part\_id].property;

var component\_driver =render.part\_information[render\_id][part\_id].component\_driver;

var component\_initialize\_data =render.part\_information[render\_id][part\_id].component\_initialize\_data;

var instance\_initialize\_data =render.part\_information[render\_id][part\_id].instance\_initialize\_data;

var part\_initialize\_data =render.part\_information[render\_id][part\_id].part\_initialize\_data;

var clip\_plane =render.clip\_plane\_array[render\_buffer\_id];

var clip\_plane\_matrix =render.clip\_plane\_matrix\_array[render\_buffer\_id];

var camera\_object\_parameter =render.camera.camera\_object\_parameter;

var camera\_render\_parameter =render.camera.camera\_render\_parameter[render\_buffer\_id];

var camera\_id =camera\_render\_parameter.camera\_id;

var camera\_component\_id =camera\_object\_parameter[camera\_id].component\_id;

var utility =render.utility;

var computer =render.computer;

var current\_time =render.current\_time;

……….

}

### Function parameters

The first parameter is method\_id. A client-side engine is a multi-pass rendering engine. Different pass draws different content. The method\_id is pass-ID, its definition is:

|  |  |  |
| --- | --- | --- |
| Method | Description | purpose |
| 0 | render scene before depth rendering | Display |
| 1 | render scene for component selection | selection |
| 2 | render scene for shadow depth | shadow |
| 3 | render depth only, no color rendering | Display |
| 4 | render scene after depth rendering | Display |
| 5 | render scene for blending | Display |
| 6 | render scene for occlusion query | Display |

The second parameter is parameter\_channel\_id, it identifies which parameter is used to render the components.

The third and fourth parameter are render\_id and part\_id of rendering part.

The fifth parameter is render\_buffer\_id. A client-side engine is a multi-target rendering engine. To improve performance client-side engine implement a buffer for each rendering target. The render\_buffer\_id is buffer IDs of rendering targets.

The six parameter component\_data\_array is an array; one element corresponds to a component needed drawn. They are created by create\_render\_parameter method of instance driver at web server, and updated by engine whenever modified.

The seventh parameter component\_part\_data is data that are created by create\_component\_render\_parameter method of part driver at web server. This parameter is data for all components. If drawn components have some shared data, you transfer it to client-side program in create\_component\_render\_parameter method of part driver, and access the shared data in the second parameter component\_part\_data.

The eighth parameter is project matrix; it has two fields: matrix that holds projection-matrix and negative\_matrix hold its inverse matrix. Usually projection-matrix is transferred to shaders on GPU, and negative\_matrix are used in transformation from view coordinate to world coordinate. Both matrix and negative\_matrix are an OpenGL-styles 16-elements array.

The ninth parameter is do\_render\_number. A client-side JavaScript function responses for all components with same render ID; once the function is called, it should draw all components with the same render and same part. This function is called for one part after another. The do\_render\_number parameter is an object with six fields. If they are zero, it means it is the first time the function is called. This parameter is used for some initialization when function is called.

1. The do\_render\_number. engine\_render is the running number of the function since engine started.
2. The do\_render\_number. engine\_part is the running number of the function for one part since engine started.
3. The do\_render\_number. render\_render is the running number of the function since the scene rendering.
4. The do\_render\_number. render\_part is the running number of the function for one part since the scene rendering.
5. The do\_render\_number. pass\_render is the running number of the function since the pass rendering.
6. The do\_render\_number. pass\_part is the running number of the function for one part since the pass rendering.

The last parameter is render. This parameter is the object that holds all scene data. All data can be found from this object.

### Render Variable

1. render.gl:

WebGL context object. You can use this object to call WebGL function.

1. render.render\_program.render\_program[render\_id].program

This variable is GPU shader program object.

1. render.buffer\_object.buffer\_object[render\_id][ part\_id]:

This variable is WebGL buffer-object object. It has four objects.

* part\_buffer\_object.face WebGL buffer object for drawing face.
* part\_buffer\_object.frame WebGL buffer object for drawing frame.
* part\_buffer\_object.edge WebGL buffer object for drawing edge.
* part\_buffer\_object.point WebGL buffer object for drawing point.

1. render.component\_location\_data

This variable holds component location, its method include:

* get\_component\_location(component\_id):get component location.
* modify\_one\_component\_location (component\_id,loca): modify component location.

1. render.data\_buffer[render\_id][part\_id]

This array holds component buffer parameter that is created by instance driver method create\_buffer\_parameter. An element of this array is also a array. Whenever component parameter is modified, server-side engine will call create\_buffer\_parameter method to transfer new parameter to client-side engine. This new parameter doesn’t replace old parameter stored in the element, whereas, the new parameter is appended to end of element array. This mechanism can help user implementing incremental data transfer.

To prevent element array becoming too large, there is a parameter max\_buffer\_data\_number defined in Part Parameter File (see configuration document); it defines maximum of element array size. If element array becomes too large, starting element of it will be deleted.

1. render.pickup:

This variable holds an object that is pinpointed by mouse. It is an object that has following field:

* render.pickup .component\_id
* render.pickup body\_id
* render.pickup face\_id
* render.pickup loop\_id
* render.pickup edge\_id
* render.pickup point\_id
* render.pickup vertex\_id
* render.pickup point\_depth
* render.pickup render\_id
* render.pickup part\_id

1. render.target

This is render target parameter. Its definition is :

[render\_purpose\_id,target\_id,render\_target\_id,component\_id,framebuffer\_width,framebuffer\_height,aspect\_value,clear\_flag,[viewport\_x0\_ratio,viewport\_y0\_ratio,viewport\_x1\_ratio,viewport\_y1\_ratio].

1. render.part\_information[render\_id][part\_id].information

This is part information. Now it has only two fields, which are information.use\_name and information.system\_name.

1. render.part\_information[render\_id][part\_id]. material

This is part material. It is created by create\_mesh\_and\_material method of part driver. This field is defined by part driver. Usually part driver put some its special data in this field.

1. render.part\_information[render\_id][part\_id]. property

This is part property. Now it has several fields.

* normal\_part\_flag: Boolean value, if true, it means this part is a normal part.
* top\_box\_flag: Boolean value, if true, it means this part is a top-box part.
* bottom\_box\_flag: Boolean value, if true, it means this part is a bottom-box part.
* part\_box:part box data. This data is an array with two elements. Each element is a three-element array(X,Y,Z).

1. render.part\_information[render\_id][part\_id].component\_driver:

This is an array indexed by buffer\_id. Every element of this array is an array with two elements. The first element is component ID, the other element is driver\_id.

1. render.part\_information[render\_id][part\_id].component\_initialize\_data:

This is an array indexed by buffer\_id. Every element of this array is a component\_initialize\_data of a component driver.

1. render.part\_information[render\_id][part\_id].instance\_initialize\_data:

This is an array indexed by buffer\_id. Every element of this array is an instance\_initialize\_data of a instance driver.

1. render.part\_information[render\_id][part\_id].part\_initialize\_data:

This is an array. Every element of this array is a part\_initialize\_data of a part driver.

1. render.clip\_plane\_array[render\_buffer\_id]:

This is clip plane parameter. It is an array [A, B, C, D]. The equation is:

Ax+By+Cz+D=0;

1. render.clip\_plane\_matrix\_array[render\_buffer\_id]:

This is clip plane projection matrix, which can transform a coordinate point onto the clip plane.

1. render.camera.camera\_object\_parameter:

This variable is an array, whose elements hold camera parameters. Every element include following fields:

* component\_id camera component\_id.
* distance camera distance.
* half\_fovy\_tanl camera half\_fovy\_tanl.
* near\_value\_ratio camera near\_value\_ratio.
* far\_value\_ratio camera far\_value\_ratio.
* projection\_matrix\_type camera projection type.

1. render.camera.camera\_render\_parameter[render\_buffer\_id]:

This data includes following fields:

* camera\_id : camera ID. The engine can have several camera. This field identifies which camera the viewer uses.
* X0,y0,dx,dy :camera view.
* mirror\_change\_matrix : transformation matrix for mirror.

1. render.utility:

This is utility function object.

1. render.computer:

This is a computation object; it can do computation such as matrix caculatation.

1. render.current\_time:

This is current time; its unit is nanosecond.

### Data storage in WebGL Buffer Object

In WebGL, all primitive(triangle, line section, point) are defined through vertex, one triangle has three vertices, one line section has two vertices, one point has one vertex. Each vertex has has several floating number, such place, normal, ID, material, texture, .etc.

Data stored in WebGL Buffer Object are vertex data for primitives, each vertex data include:

1. vertex coordinate: four floating-number, three for Vertex place(X Y Z), the last one is usually 1.0. You can also define some other values in your mesh file on server. You can even defined several floating-number for each vertex and put them here.
2. normal coordinate: four floating-number, three for Vertex normal(DX, DY, DZ), the last one is usually 1.0. You can also define some other values in your mesh file on server. You can even defined several floating-number for each vertex and put them here.
3. vertex IDs: four floating-number. body ID, face ID, vertex ID, the last value is primitive type, 0 for face, 1 for edge, 2 for point, .etc.
4. material: four floating-number, all from mesh file on server. No explanation.
5. attribute coordinate: each attribute has four floating-number(x, y, z, w), similar format to vertex coordinate and normal coordinate.

## Predefined variable in shaders

### System information variable

uniform system\_information

{

float pickup\_depth;

float pickup\_value;

float canvas\_aspect\_value;

int pickup\_component\_id;

int pickup\_render\_id;

int pickup\_part\_id;

int pickup\_body\_id;

int pickup\_face\_id;

int pickup\_vertex\_id;

int pickup\_loop\_id;

int pickup\_edge\_id;

int pickup\_point\_id;

int year;

int month;

int day;

int hour;

int minute;

int second;

int millisecond;

int microsecond;

int nanosecond;

int canvas\_width;

int canvas\_height;

}system\_info;

### Target information variable

uniform target\_information

{

mat4 projection\_matrix;

mat4 negative\_projection\_matrix;

mat4 other\_projection\_matrix;

mat4 other\_negative\_projection\_matrix;

mat4 screen\_move\_matrix;

mat4 camera\_location;

mat4 clip\_plane\_matrix;

vec4 left\_plane;

vec4 right\_plane;

vec4 up\_plane;

vec4 down\_plane;

vec4 near\_plane;

vec4 far\_plane;

vec4 center\_plane;

vec4 clip\_plane;

vec4 far\_center\_point;

vec4 center\_point;

vec4 near\_center\_point;

vec4 eye\_point;

vec4 left\_down\_near\_point;

vec4 left\_up\_near\_point;

vec4 right\_down\_near\_point;

vec4 right\_up\_near\_point;

vec4 left\_down\_center\_point;

vec4 left\_up\_center\_point;

vec4 right\_down\_center\_point;

vec4 right\_up\_center\_point;

vec4 left\_down\_far\_point;

vec4 left\_up\_far\_point;

vec4 right\_down\_far\_point;

vec4 right\_up\_far\_point;

float target\_aspect\_value;

float half\_fovy\_tanl;

float near\_value\_ratio;

float far\_value\_ratio;

float distance;

float near\_value;

float far\_value;

float center\_point\_depth;

int projection\_type\_flag;

int target\_width;

int target\_height;

int draw\_buffer\_id;

}target\_info;

### Pass information variable

uniform pass\_information

{

vec4 clear\_color;

int method\_id;

int clear\_flag;

ivec4 viewport;

}pass\_info;

### Component information variable

uniform component\_information

{

mat4 model\_matrix;

int component\_id;

}component\_info;

## Predefined functions in shaders

vec4 code\_integer(int my\_id)

{

float step =128.0;

float dlt =0.25;

float color =float(my\_id+1)+dlt;

float red\_color,green\_color,blue\_color,alf\_color;

color=color/step; alf\_color =fract(color); color=floor(color)+dlt;

color=color/step; blue\_color =fract(color); color=floor(color)+dlt;

color=color/step; green\_color=fract(color); color=floor(color)+dlt;

color=color/step; red\_color =fract(color); color=floor(color)+dlt;

return vec4(red\_color/2.0,green\_color/2.0,blue\_color/2.0,alf\_color/2.0);

}

vec4 code\_float(float value)

{

float x,y,z,alf;

value=(value<-1.0)?-1.0:(value>1.0)?1.0:value;

value=(value+1.0)/2.0;

value=128.0\*value;

alf=(floor(value)+0.01)/256.0;

value=fract(value);

value=128.0\*value;

z=(floor(value)+0.01)/256.0;

value=fract(value);

value=128.0\*value;

y=(floor(value)+0.01)/256.0;

value=fract(value);

value=128.0\*value;

x=(floor(value)+0.01)/256.0;

value=fract(value);

return vec4(x,y,z,alf);

}