if (\_runningScene)

{

#if (CC\_USE\_PHYSICS || (CC\_USE\_3D\_PHYSICS && CC\_ENABLE\_BULLET\_INTEGRATION) || CC\_USE\_NAVMESH)

\_runningScene->stepPhysicsAndNavigation(\_deltaTime);

#endif

//clear draw stats

\_renderer->clearDrawStats();

//render the scene

**\_runningScene->render(\_renderer);**

\_eventDispatcher->dispatchEvent(\_eventAfterVisit);

}

**void Scene::render(Renderer\* renderer)**

{

auto director = Director::getInstance();

Camera\* defaultCamera = nullptr;

const auto& transform = getNodeToParentTransform();

for (const auto& camera : getCameras())

{

if (!camera->isVisible())

continue;

Camera::\_visitingCamera = camera;

if (Camera::\_visitingCamera->getCameraFlag() == CameraFlag::DEFAULT)

{

defaultCamera = Camera::\_visitingCamera;

}

director->pushMatrix(MATRIX\_STACK\_TYPE::MATRIX\_STACK\_PROJECTION);

director->loadMatrix(MATRIX\_STACK\_TYPE::MATRIX\_STACK\_PROJECTION, Camera::\_visitingCamera->getViewProjectionMatrix());

camera->apply();

//clear background with max depth

camera->clearBackground();

//visit the scene

**visit(renderer, transform, 0);**

#if CC\_USE\_NAVMESH

if (\_navMesh && \_navMeshDebugCamera == camera)

{

\_navMesh->debugDraw(renderer);

}

#endif

**renderer->render();**

director->popMatrix(MATRIX\_STACK\_TYPE::MATRIX\_STACK\_PROJECTION);

}

#if CC\_USE\_3D\_PHYSICS && CC\_ENABLE\_BULLET\_INTEGRATION

if (\_physics3DWorld && \_physics3DWorld->isDebugDrawEnabled())

{

director->pushMatrix(MATRIX\_STACK\_TYPE::MATRIX\_STACK\_PROJECTION);

director->loadMatrix(MATRIX\_STACK\_TYPE::MATRIX\_STACK\_PROJECTION, \_physics3dDebugCamera != nullptr ? \_physics3dDebugCamera->getViewProjectionMatrix() : defaultCamera->getViewProjectionMatrix());

\_physics3DWorld->debugDraw(renderer);

renderer->render();

director->popMatrix(MATRIX\_STACK\_TYPE::MATRIX\_STACK\_PROJECTION);

}

#endif

Camera::\_visitingCamera = nullptr;

experimental::FrameBuffer::applyDefaultFBO();

}

**void Node::visit(Renderer\* renderer, const Mat4 &parentTransform, uint32\_t parentFlags)**

{

// quick return if not visible. children won't be drawn.

if (!\_visible)

{

return;

}

uint32\_t flags = processParentFlags(parentTransform, parentFlags);

// IMPORTANT:

// To ease the migration to v3.0, we still support the Mat4 stack,

// but it is deprecated and your code should not rely on it

\_director->pushMatrix(MATRIX\_STACK\_TYPE::MATRIX\_STACK\_MODELVIEW);

\_director->loadMatrix(MATRIX\_STACK\_TYPE::MATRIX\_STACK\_MODELVIEW, \_modelViewTransform);

bool visibleByCamera = isVisitableByVisitingCamera();

int i = 0;

if(!\_children.empty())

{

**sortAllChildren();**

// draw children zOrder < 0

for( ; i < \_children.size(); i++ )

{

auto node = \_children.at(i);

if (node && node->\_localZOrder < 0)

node->visit(renderer, \_modelViewTransform, flags);

else

break;

}

// self draw

if (visibleByCamera)

**this->draw(renderer, \_modelViewTransform, flags);**

for(auto it=\_children.cbegin()+i; it != \_children.cend(); ++it)

**(\*it)->visit(renderer, \_modelViewTransform, flags);**

}

else if (visibleByCamera)

{

**this->draw(renderer, \_modelViewTransform, flags);**

}

\_director->popMatrix(MATRIX\_STACK\_TYPE::MATRIX\_STACK\_MODELVIEW);

// FIX ME: Why need to set \_orderOfArrival to 0??

// Please refer to https://github.com/cocos2d/cocos2d-x/pull/6920

// reset for next frame

// \_orderOfArrival = 0;

}

**注意上面的sortAllChildren函数：**

void Node::sortAllChildren()

{

if (\_reorderChildDirty)

{

std::sort(std::begin(\_children), std::end(\_children), nodeComparisonLess);

\_reorderChildDirty = false;

}

}

bool nodeComparisonLess(Node\* n1, Node\* n2)

{

**return( n1->getLocalZOrder() < n2->getLocalZOrder() ||**

**( n1->getLocalZOrder() == n2->getLocalZOrder() && n1->getOrderOfArrival() < n2->getOrderOfArrival() )**

**);**

}

也就是先通过localZOrder进行排序，如果localZOrder相等，则按照OrderOfArrival进行排序，OrderOfArrival表示来到的顺序，这个序号在addChild的时候进行了顺序递增，也就是先来的序号小，后来的序号大

void Node::addChildHelper(Node\* child, int localZOrder, int tag, const std::string &name, bool setTag)

{

if (\_children.empty())

{

this->childrenAlloc();

}

this->insertChild(child, localZOrder);

if (setTag)

child->setTag(tag);

else

child->setName(name);

child->setParent(this);

**child->setOrderOfArrival(s\_globalOrderOfArrival++);**

if( \_running )

{

child->onEnter();

// prevent onEnterTransitionDidFinish to be called twice when a node is added in onEnter

if (\_isTransitionFinished)

{

child->onEnterTransitionDidFinish();

}

}

if (\_cascadeColorEnabled)

{

updateCascadeColor();

}

if (\_cascadeOpacityEnabled)

{

updateCascadeOpacity();

}

}

比如Spirte

V3F\_C4B\_T2F\_Quad \_quad;

struct CC\_DLL V3F\_C4B\_T2F\_Quad

{

/// top left

V3F\_C4B\_T2F tl;

/// bottom left

V3F\_C4B\_T2F bl;

/// top right

V3F\_C4B\_T2F tr;

/// bottom right

V3F\_C4B\_T2F br;

};

struct CC\_DLL V3F\_C4B\_T2F

{

/// vertices (3F)

Vec3 vertices; // 12 bytes

/// colors (4B)

Color4B colors; // 4 bytes

// tex coords (2F)

Tex2F texCoords; // 8 bytes

};

在初始化的时候会调用：

bool Sprite::initWithTexture(Texture2D \*texture, const Rect& rect, bool rotated)

{

bool result;

if (Node::init())

{

\_batchNode = nullptr;

\_recursiveDirty = false;

setDirty(false);

\_opacityModifyRGB = true;

\_blendFunc = BlendFunc::ALPHA\_PREMULTIPLIED;

\_flippedX = \_flippedY = false;

// default transform anchor: center

setAnchorPoint(Vec2(0.5f, 0.5f));

// zwoptex default values

\_offsetPosition.setZero();

// clean the Quad

memset(&\_quad, 0, sizeof(\_quad));

**// Atlas: Color // 填充顶点颜色**

**\_quad.bl.colors = Color4B::WHITE;**

**\_quad.br.colors = Color4B::WHITE;**

**\_quad.tl.colors = Color4B::WHITE;**

**\_quad.tr.colors = Color4B::WHITE;**

// shader state

setGLProgramState(GLProgramState::getOrCreateWithGLProgramName(GLProgram::SHADER\_NAME\_POSITION\_TEXTURE\_COLOR\_NO\_MVP));

// update texture (calls updateBlendFunc)

setTexture(texture);

**setTextureRect(rect, rotated, rect.size);**

// by default use "Self Render".

// if the sprite is added to a batchnode, then it will automatically switch to "batchnode Render"

setBatchNode(nullptr);

result = true;

}

else

{

result = false;

}

\_recursiveDirty = true;

setDirty(true);

return result;

}

void Sprite::setTextureRect(const Rect& rect, bool rotated, const Size& untrimmedSize)

{

\_rectRotated = rotated;

setContentSize(untrimmedSize);

setVertexRect(rect);

**setTextureCoords(rect);**

float relativeOffsetX = \_unflippedOffsetPositionFromCenter.x;

float relativeOffsetY = \_unflippedOffsetPositionFromCenter.y;

// issue #732

if (\_flippedX)

{

relativeOffsetX = -relativeOffsetX;

}

if (\_flippedY)

{

relativeOffsetY = -relativeOffsetY;

}

\_offsetPosition.x = relativeOffsetX + (\_contentSize.width - \_rect.size.width) / 2;

\_offsetPosition.y = relativeOffsetY + (\_contentSize.height - \_rect.size.height) / 2;

// rendering using batch node

if (\_batchNode)

{

// update dirty\_, don't update recursiveDirty\_

setDirty(true);

}

else

{

// self rendering

// Atlas: Vertex

float x1 = 0.0f + \_offsetPosition.x;

float y1 = 0.0f + \_offsetPosition.y;

float x2 = x1 + \_rect.size.width;

float y2 = y1 + \_rect.size.height;

**// Don't update Z. // 填充顶点位置**

**\_quad.bl.vertices.set(x1, y1, 0.0f);**

**\_quad.br.vertices.set(x2, y1, 0.0f);**

**\_quad.tl.vertices.set(x1, y2, 0.0f);**

**\_quad.tr.vertices.set(x2, y2, 0.0f);**

}

**\_polyInfo.setQuad(&\_quad); // 将信息保存到polyInfo中**

}

void Sprite::setTextureCoords(Rect rect)

{

rect = CC\_RECT\_POINTS\_TO\_PIXELS(rect);

Texture2D \*tex = \_batchNode ? \_textureAtlas->getTexture() : \_texture;

if (! tex)

{

return;

}

float atlasWidth = (float)tex->getPixelsWide();

float atlasHeight = (float)tex->getPixelsHigh();

float left, right, top, bottom;

if (\_rectRotated)

{

#if CC\_FIX\_ARTIFACTS\_BY\_STRECHING\_TEXEL

left = (2\*rect.origin.x+1)/(2\*atlasWidth);

right = left+(rect.size.height\*2-2)/(2\*atlasWidth);

top = (2\*rect.origin.y+1)/(2\*atlasHeight);

bottom = top+(rect.size.width\*2-2)/(2\*atlasHeight);

#else

left = rect.origin.x/atlasWidth;

right = (rect.origin.x+rect.size.height) / atlasWidth;

top = rect.origin.y/atlasHeight;

bottom = (rect.origin.y+rect.size.width) / atlasHeight;

#endif // CC\_FIX\_ARTIFACTS\_BY\_STRECHING\_TEXEL

if (\_flippedX)

{

std::swap(top, bottom);

}

if (\_flippedY)

{

std::swap(left, right);

}

\_quad.bl.texCoords.u = left;

\_quad.bl.texCoords.v = top;

\_quad.br.texCoords.u = left;

\_quad.br.texCoords.v = bottom;

\_quad.tl.texCoords.u = right;

\_quad.tl.texCoords.v = top;

\_quad.tr.texCoords.u = right;

\_quad.tr.texCoords.v = bottom;

}

else

{

#if CC\_FIX\_ARTIFACTS\_BY\_STRECHING\_TEXEL

left = (2\*rect.origin.x+1)/(2\*atlasWidth);

right = left + (rect.size.width\*2-2)/(2\*atlasWidth);

top = (2\*rect.origin.y+1)/(2\*atlasHeight);

bottom = top + (rect.size.height\*2-2)/(2\*atlasHeight);

#else

left = rect.origin.x/atlasWidth;

right = (rect.origin.x + rect.size.width) / atlasWidth;

top = rect.origin.y/atlasHeight;

bottom = (rect.origin.y + rect.size.height) / atlasHeight;

#endif // ! CC\_FIX\_ARTIFACTS\_BY\_STRECHING\_TEXEL

if(\_flippedX)

{

std::swap(left, right);

}

if(\_flippedY)

{

std::swap(top, bottom);

}

**// 填充顶点纹理**

**\_quad.bl.texCoords.u = left;**

**\_quad.bl.texCoords.v = bottom;**

**\_quad.br.texCoords.u = right;**

**\_quad.br.texCoords.v = bottom;**

**\_quad.tl.texCoords.u = left;**

**\_quad.tl.texCoords.v = top;**

**\_quad.tr.texCoords.u = right;**

**\_quad.tr.texCoords.v = top;**

}

}

**Draw函数**

void Sprite::draw(Renderer \*renderer, const Mat4 &transform, uint32\_t flags)

{

#if CC\_USE\_CULLING

// Don't do calculate the culling if the transform was not updated

auto visitingCamera = Camera::getVisitingCamera();

auto defaultCamera = Camera::getDefaultCamera();

if (visitingCamera == defaultCamera) {

\_insideBounds = ((flags & FLAGS\_TRANSFORM\_DIRTY)|| visitingCamera->isViewProjectionUpdated()) ? renderer->checkVisibility(transform, \_contentSize) : \_insideBounds;

}

else

{

\_insideBounds = renderer->checkVisibility(transform, \_contentSize);

}

if(\_insideBounds)

#endif

{

**\_trianglesCommand.init(\_globalZOrder, \_texture->getName(), getGLProgramState(), \_blendFunc, \_polyInfo.triangles, transform, flags);**

**renderer->addCommand(&\_trianglesCommand);**

#if CC\_SPRITE\_DEBUG\_DRAW

\_debugDrawNode->clear();

auto count = \_polyInfo.triangles.indexCount/3;

auto indices = \_polyInfo.triangles.indices;

auto verts = \_polyInfo.triangles.verts;

for(ssize\_t i = 0; i < count; i++)

{

//draw 3 lines

Vec3 from =verts[indices[i\*3]].vertices;

Vec3 to = verts[indices[i\*3+1]].vertices;

\_debugDrawNode->drawLine(Vec2(from.x, from.y), Vec2(to.x,to.y), Color4F::WHITE);

from =verts[indices[i\*3+1]].vertices;

to = verts[indices[i\*3+2]].vertices;

\_debugDrawNode->drawLine(Vec2(from.x, from.y), Vec2(to.x,to.y), Color4F::WHITE);

from =verts[indices[i\*3+2]].vertices;

to = verts[indices[i\*3]].vertices;

\_debugDrawNode->drawLine(Vec2(from.x, from.y), Vec2(to.x,to.y), Color4F::WHITE);

}

#endif //CC\_SPRITE\_DEBUG\_DRAW

}

}

/\*\*

\* PolygonInfo is an object holding the required data to display Sprites.

\* It can be a simple as a triangle, or as complex as a whole 3D mesh

\*/

class CC\_DLL PolygonInfo

{

public:

/// @name Creators

/// @{

/\*\*

\* Creates an empty Polygon info

\* @memberof PolygonInfo

\* @return PolygonInfo object

\*/

PolygonInfo():

isVertsOwner(true),

rect(cocos2d::Rect::ZERO),

filename("")

{

triangles.verts = nullptr;

triangles.indices = nullptr;

triangles.vertCount = 0;

triangles.indexCount = 0;

};

/\*\*

\* Create an polygoninfo from the data of another Polygoninfo

\* @param other another PolygonInfo to be copied

\* @return duplicate of the other PolygonInfo

\*/

PolygonInfo(const PolygonInfo& other);

// end of creators group

/// @}

/\*\*

\* Copy the member of the other PolygonInfo

\* @param other another PolygonInfo to be copied

\*/

PolygonInfo& operator= (const PolygonInfo &other);

~PolygonInfo();

/\*\*

\* set the data to be a pointer to a quad

\* the member verts will not be released when this PolygonInfo destructs

\* as the verts memory are managed by other objects

\* @param quad a pointer to the V3F\_C4B\_T2F\_Quad object

\*/

**void setQuad(V3F\_C4B\_T2F\_Quad \*quad);**

/\*\*

\* set the data to be a pointer to a triangles

\* the member verts will not be released when this PolygonInfo destructs

\* as the verts memory are managed by other objects

\* @param triangles a pointer to the TrianglesCommand::Triangles object

\*/

void setTriangles(TrianglesCommand::Triangles triangles);

/\*\*

\* get vertex count

\* @return number of vertices

\*/

const unsigned int getVertCount() const;

/\*\*

\* get triangles count

\* @return number of triangles

\*/

const unsigned int getTriaglesCount() const;

/\*\*

\* get sum of all triangle area size

\* @return sum of all triangle area size

\*/

const float getArea() const;

Rect rect;

std::string filename;

TrianglesCommand::Triangles triangles;

protected:

bool isVertsOwner;

private:

void releaseVertsAndIndices();

};

**void PolygonInfo::setQuad(V3F\_C4B\_T2F\_Quad \*quad)**

{

releaseVertsAndIndices();

isVertsOwner = false;

triangles.indices = quadIndices;

triangles.vertCount = 4;

triangles.indexCount = 6;

triangles.verts = (V3F\_C4B\_T2F\*)quad;

}

**static unsigned short quadIndices[]={0,1,2, 3,2,1};**

**最后查看Renderer：**

**std::stack<int> \_commandGroupStack;**

**std::vector<RenderQueue> \_renderGroups;**

**//for TrianglesCommand**

**V3F\_C4B\_T2F \_verts[VBO\_SIZE]; //VBO\_SIZE=65536**

**GLushort \_indices[INDEX\_VBO\_SIZE]; //INDEX\_VBO\_SIZE=65536\*6/4**

**GLuint \_buffersVAO;**

**GLuint \_buffersVBO[2]; //0: vertex 1: indices**

**int \_filledVertex;**

**int \_filledIndex;**

**//for QuadCommand**

**V3F\_C4B\_T2F \_quadVerts[VBO\_SIZE];**

**GLushort \_quadIndices[INDEX\_VBO\_SIZE];**

**GLuint \_quadVAO;**

**GLuint \_quadbuffersVBO[2]; //0: vertex 1: indices**

**int \_numberQuads;**

**在Render的构造函数中默认构造了一个RenderaQueue并将其加入到了renderGroup中，同时\_commandGroupStack中的默认序号为0**

**看看RenderQueue**

class RenderQueue {

public:

/\*\*

RenderCommand will be divided into Queue Groups.

\*/

enum QUEUE\_GROUP

{

**/\*\*Objects with globalZ smaller than 0.\*/**

**GLOBALZ\_NEG = 0, // global<0的2D物体**

**/\*\*Opaque 3D objects with 0 globalZ.\*/**

**OPAQUE\_3D = 1, // 不透明的3D物体**

**/\*\*Transparent 3D objects with 0 globalZ.\*/**

**TRANSPARENT\_3D = 2, // 透明的3D物体**

**/\*\*2D objects with 0 globalZ.\*/**

**GLOBALZ\_ZERO = 3, // global=0的2D物体**

**/\*\*Objects with globalZ bigger than 0.\*/**

**GLOBALZ\_POS = 4, // global>0的2D物体**

**QUEUE\_COUNT = 5,**

};

public:

/\*\*Constructor.\*/

RenderQueue();

/\*\*Push a renderCommand into current renderqueue.\*/

**void push\_back(RenderCommand\* command);**

/\*\*Return the number of render commands.\*/

ssize\_t size() const;

/\*\*Sort the render commands.\*/

void sort();

/\*\*Treat sorted commands as an array, access them one by one.\*/

RenderCommand\* operator[](ssize\_t index) const;

/\*\*Clear all rendered commands.\*/

void clear();

/\*\*Realloc command queues and reserve with given size. Note: this clears any existing commands.\*/

void realloc(size\_t reserveSize);

/\*\*Get a sub group of the render queue.\*/

inline std::vector<RenderCommand\*>& getSubQueue(QUEUE\_GROUP group) { return \_commands[group]; }

/\*\*Get the number of render commands contained in a subqueue.\*/

inline ssize\_t getSubQueueSize(QUEUE\_GROUP group) const { return \_commands[group].size();}

/\*\*Save the current DepthState, CullState, DepthWriteState render state.\*/

void saveRenderState();

/\*\*Restore the saved DepthState, CullState, DepthWriteState render state.\*/

void restoreRenderState();

protected:

/\*\*The commands in the render queue.\*/

**std::vector<RenderCommand\*> \_commands[QUEUE\_COUNT];**

/\*\*Cull state.\*/

bool \_isCullEnabled;

/\*\*Depth test enable state.\*/

bool \_isDepthEnabled;

/\*\*Depth buffer write state.\*/

GLboolean \_isDepthWrite;

};

看看RenderQueue中的push\_back实际上是根据renderCommand的类型分别存入不同的RenderCommand数组中（实际上是vector）：

**void RenderQueue::push\_back(RenderCommand\* command)**

{

float z = command->getGlobalOrder();

if(z < 0)

{

\_commands[QUEUE\_GROUP::GLOBALZ\_NEG].push\_back(command);

}

else if(z > 0)

{

\_commands[QUEUE\_GROUP::GLOBALZ\_POS].push\_back(command);

}

else

{

if(command->is3D())

{

if(command->isTransparent())

{

\_commands[QUEUE\_GROUP::TRANSPARENT\_3D].push\_back(command);

}

else

{

\_commands[QUEUE\_GROUP::OPAQUE\_3D].push\_back(command);

}

}

else

{

\_commands[QUEUE\_GROUP::GLOBALZ\_ZERO].push\_back(command);

}

}

}

void Renderer::addCommand(RenderCommand\* command)

{

int renderQueue =\_commandGroupStack.top();

**addCommand(command, renderQueue);**

}

void Renderer::addCommand(RenderCommand\* command, int renderQueue)

{

CCASSERT(!\_isRendering, "Cannot add command while rendering");

CCASSERT(renderQueue >=0, "Invalid render queue");

CCASSERT(command->getType() != RenderCommand::Type::UNKNOWN\_COMMAND, "Invalid Command Type");

**\_renderGroups[renderQueue].push\_back(command);**

}

Render内部保存了一个命令的绘制队列，cocos2dx的所有绘制都会被封装成为命令发送到当前的绘制队列中。

当整个子节点遍历后，即所有的绘制命令加入到绘制队列上，这里调用render->render()，这个函数的作用是绘制当前绘制队列中的绘制信息，最终是调用opengl的命令完成绘制，**最终实际上是调用flush函数实现绘制的**

void Renderer::render()

{

//Uncomment this once everything is rendered by new renderer

//glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

//TODO: setup camera or MVP

\_isRendering = true;

if (\_glViewAssigned)

{

//Process render commands

//1. Sort render commands based on ID

for (auto &renderqueue : \_renderGroups)

{

renderqueue.sort();

}

**visitRenderQueue(\_renderGroups[0]);**

}

clean();

\_isRendering = false;

}

void Renderer::visitRenderQueue(RenderQueue& queue)

{

queue.saveRenderState();

//

//Process Global-Z < 0 Objects

//

const auto& zNegQueue = queue.getSubQueue(RenderQueue::QUEUE\_GROUP::GLOBALZ\_NEG);

if (zNegQueue.size() > 0)

{

if(\_isDepthTestFor2D)

{

glEnable(GL\_DEPTH\_TEST);

glDepthMask(true);

glEnable(GL\_BLEND);

RenderState::StateBlock::\_defaultState->setDepthTest(true);

RenderState::StateBlock::\_defaultState->setDepthWrite(true);

RenderState::StateBlock::\_defaultState->setBlend(true);

}

else

{

glDisable(GL\_DEPTH\_TEST);

glDepthMask(false);

glEnable(GL\_BLEND);

RenderState::StateBlock::\_defaultState->setDepthTest(false);

RenderState::StateBlock::\_defaultState->setDepthWrite(false);

RenderState::StateBlock::\_defaultState->setBlend(true);

}

glDisable(GL\_CULL\_FACE);

RenderState::StateBlock::\_defaultState->setCullFace(false);

for (auto it = zNegQueue.cbegin(); it != zNegQueue.cend(); ++it)

{

processRenderCommand(\*it);

}

**flush();**

}

//

//Process Opaque Object

//

const auto& opaqueQueue = queue.getSubQueue(RenderQueue::QUEUE\_GROUP::OPAQUE\_3D);

if (opaqueQueue.size() > 0)

{

//Clear depth to achieve layered rendering

glEnable(GL\_DEPTH\_TEST);

glDepthMask(true);

glDisable(GL\_BLEND);

glEnable(GL\_CULL\_FACE);

RenderState::StateBlock::\_defaultState->setDepthTest(true);

RenderState::StateBlock::\_defaultState->setDepthWrite(true);

RenderState::StateBlock::\_defaultState->setBlend(false);

RenderState::StateBlock::\_defaultState->setCullFace(true);

for (auto it = opaqueQueue.cbegin(); it != opaqueQueue.cend(); ++it)

{

processRenderCommand(\*it);

}

**flush();**

}

//

//Process 3D Transparent object

//

const auto& transQueue = queue.getSubQueue(RenderQueue::QUEUE\_GROUP::TRANSPARENT\_3D);

if (transQueue.size() > 0)

{

glEnable(GL\_DEPTH\_TEST);

glDepthMask(false);

glEnable(GL\_BLEND);

glEnable(GL\_CULL\_FACE);

RenderState::StateBlock::\_defaultState->setDepthTest(true);

RenderState::StateBlock::\_defaultState->setDepthWrite(false);

RenderState::StateBlock::\_defaultState->setBlend(true);

RenderState::StateBlock::\_defaultState->setCullFace(true);

for (auto it = transQueue.cbegin(); it != transQueue.cend(); ++it)

{

processRenderCommand(\*it);

}

**flush();**

}

//

//Process Global-Z = 0 Queue

//

const auto& zZeroQueue = queue.getSubQueue(RenderQueue::QUEUE\_GROUP::GLOBALZ\_ZERO);

if (zZeroQueue.size() > 0)

{

if(\_isDepthTestFor2D)

{

glEnable(GL\_DEPTH\_TEST);

glDepthMask(true);

glEnable(GL\_BLEND);

RenderState::StateBlock::\_defaultState->setDepthTest(true);

RenderState::StateBlock::\_defaultState->setDepthWrite(true);

RenderState::StateBlock::\_defaultState->setBlend(true);

}

else

{

glDisable(GL\_DEPTH\_TEST);

glDepthMask(false);

glEnable(GL\_BLEND);

RenderState::StateBlock::\_defaultState->setDepthTest(false);

RenderState::StateBlock::\_defaultState->setDepthWrite(false);

RenderState::StateBlock::\_defaultState->setBlend(true);

}

glDisable(GL\_CULL\_FACE);

RenderState::StateBlock::\_defaultState->setCullFace(false);

for (auto it = zZeroQueue.cbegin(); it != zZeroQueue.cend(); ++it)

{

processRenderCommand(\*it);

}

**flush();**

}

//

//Process Global-Z > 0 Queue

//

const auto& zPosQueue = queue.getSubQueue(RenderQueue::QUEUE\_GROUP::GLOBALZ\_POS);

if (zPosQueue.size() > 0)

{

if(\_isDepthTestFor2D)

{

glEnable(GL\_DEPTH\_TEST);

glDepthMask(true);

glEnable(GL\_BLEND);

RenderState::StateBlock::\_defaultState->setDepthTest(true);

RenderState::StateBlock::\_defaultState->setDepthWrite(true);

RenderState::StateBlock::\_defaultState->setBlend(true);

}

else

{

glDisable(GL\_DEPTH\_TEST);

glDepthMask(false);

glEnable(GL\_BLEND);

RenderState::StateBlock::\_defaultState->setDepthTest(false);

RenderState::StateBlock::\_defaultState->setDepthWrite(false);

RenderState::StateBlock::\_defaultState->setBlend(true);

}

glDisable(GL\_CULL\_FACE);

RenderState::StateBlock::\_defaultState->setCullFace(false);

for (auto it = zPosQueue.cbegin(); it != zPosQueue.cend(); ++it)

{

**processRenderCommand(\*it);**

}

**flush();**

}

queue.restoreRenderState();

}

**void Renderer::processRenderCommand(RenderCommand\* command)**

{

auto commandType = command->getType();

if( RenderCommand::Type::TRIANGLES\_COMMAND == commandType)

{

//Draw if we have batched other commands which are not triangle command

**// 如果遇到一个三角形命令，则将之前储存3D与四边形命令立即渲染，这样可以保证后面\_batchedCommands中添加的三角形始终是一段连续的三角形命令**

flush3D();

flushQuads();

//Process triangle command

auto cmd = static\_cast<TrianglesCommand\*>(command);

//Draw batched Triangles if necessary

if(cmd->isSkipBatching() || \_filledVertex + cmd->getVertexCount() > VBO\_SIZE || \_filledIndex + cmd->getIndexCount() > INDEX\_VBO\_SIZE)

{

CCASSERT(cmd->getVertexCount()>= 0 && cmd->getVertexCount() < VBO\_SIZE, "VBO for vertex is not big enough, please break the data down or use customized render command");

CCASSERT(cmd->getIndexCount()>= 0 && cmd->getIndexCount() < INDEX\_VBO\_SIZE, "VBO for index is not big enough, please break the data down or use customized render command");

//Draw batched Triangles if VBO is full

drawBatchedTriangles();

}

//Batch Triangles

**\_batchedCommands.push\_back(cmd); // 将三角形Command放入三角形bath中**

**fillVerticesAndIndices(cmd); // 将该三角形填充入vbo**

if(cmd->isSkipBatching())

{

drawBatchedTriangles();

}

}

else if ( RenderCommand::Type::QUAD\_COMMAND == commandType )

{

//Draw if we have batched other commands which are not quad command

**// 如果遇到一个四边形命令，则将之前储存3D与三角形命令立即渲染，这样可以保证后面\_batchQuadCommands中添加的三角形始终是一段连续的四边形命令**

flush3D();

flushTriangles();

//Process quad command

auto cmd = static\_cast<QuadCommand\*>(command);

//Draw batched quads if necessary

if(cmd->isSkipBatching()|| (\_numberQuads + cmd->getQuadCount()) \* 4 > VBO\_SIZE )

{

CCASSERT(cmd->getQuadCount()>= 0 && cmd->getQuadCount() \* 4 < VBO\_SIZE, "VBO for vertex is not big enough, please break the data down or use customized render command");

//Draw batched quads if VBO is full

drawBatchedQuads();

}

//Batch Quads

**\_batchQuadCommands.push\_back(cmd);** **// 将四边形Command放入四边形bath中**

**fillQuads(cmd);**

if(cmd->isSkipBatching())

{

drawBatchedQuads();

}

}

else if (RenderCommand::Type::MESH\_COMMAND == commandType)

{

flush2D();

auto cmd = static\_cast<MeshCommand\*>(command);

if (cmd->isSkipBatching() || \_lastBatchedMeshCommand == nullptr || \_lastBatchedMeshCommand->getMaterialID() != cmd->getMaterialID())

{

flush3D();

if(cmd->isSkipBatching())

{

// XXX: execute() will call bind() and unbind()

// but unbind() shouldn't be call if the next command is a MESH\_COMMAND with Material.

// Once most of cocos2d-x moves to Pass/StateBlock, only bind() should be used.

cmd->execute();

}

else

{

cmd->preBatchDraw();

cmd->batchDraw();

\_lastBatchedMeshCommand = cmd;

}

}

else

{

cmd->batchDraw();

}

}

else if(RenderCommand::Type::GROUP\_COMMAND == commandType)

{

flush();

int renderQueueID = ((GroupCommand\*) command)->getRenderQueueID();

visitRenderQueue(\_renderGroups[renderQueueID]);

}

else if(RenderCommand::Type::CUSTOM\_COMMAND == commandType)

{

flush();

auto cmd = static\_cast<CustomCommand\*>(command);

cmd->execute();

}

else if(RenderCommand::Type::BATCH\_COMMAND == commandType)

{

flush();

auto cmd = static\_cast<BatchCommand\*>(command);

cmd->execute();

}

else if(RenderCommand::Type::PRIMITIVE\_COMMAND == commandType)

{

flush();

auto cmd = static\_cast<PrimitiveCommand\*>(command);

cmd->execute();

}

else

{

CCLOGERROR("Unknown commands in renderQueue");

}

}

void Renderer::flush()

{

**flush2D();**

**flush3D();**

}

void Renderer::flush2D()

{

**flushQuads();**

**flushTriangles();**

}

void Renderer::flush3D()

{

if (\_lastBatchedMeshCommand)

{

\_lastBatchedMeshCommand->postBatchDraw();

\_lastBatchedMeshCommand = nullptr;

}

}

**void Renderer::flushTriangles()**

{

if(\_filledIndex > 0)

{

**drawBatchedTriangles();**

\_lastMaterialID = 0;

}

}

**void Renderer::drawBatchedTriangles()**

{

//TODO: we can improve the draw performance by insert material switching command before hand.

int indexToDraw = 0;

int startIndex = 0;

//Upload buffer to VBO

if(\_filledVertex <= 0 || \_filledIndex <= 0 || \_batchedCommands.empty())

{

return;

}

if (Configuration::getInstance()->supportsShareableVAO())

{

//Bind VAO

GL::bindVAO(\_buffersVAO);

//Set VBO data

glBindBuffer(GL\_ARRAY\_BUFFER, \_buffersVBO[0]);

// option 1: subdata

// glBufferSubData(GL\_ARRAY\_BUFFER, sizeof(\_quads[0])\*start, sizeof(\_quads[0]) \* n , &\_quads[start] );

// option 2: data

// glBufferData(GL\_ARRAY\_BUFFER, sizeof(quads\_[0]) \* (n-start), &quads\_[start], GL\_DYNAMIC\_DRAW);

// option 3: orphaning + glMapBuffer

glBufferData(GL\_ARRAY\_BUFFER, sizeof(\_verts[0]) \* \_filledVertex, nullptr, **GL\_DYNAMIC\_DRAW**);

void \*buf = glMapBuffer(GL\_ARRAY\_BUFFER, GL\_WRITE\_ONLY);

memcpy(buf, \_verts, sizeof(\_verts[0])\* \_filledVertex);

glUnmapBuffer(GL\_ARRAY\_BUFFER);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, \_buffersVBO[1]);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(\_indices[0]) \* \_filledIndex, \_indices, **GL\_STATIC\_DRAW**); // 注意索引为静态，不要改变

}

else

{

#define kQuadSize sizeof(\_verts[0])

glBindBuffer(GL\_ARRAY\_BUFFER, \_buffersVBO[0]);

glBufferData(GL\_ARRAY\_BUFFER, sizeof(\_verts[0]) \* \_filledVertex , \_verts, GL\_DYNAMIC\_DRAW);

GL::enableVertexAttribs(GL::VERTEX\_ATTRIB\_FLAG\_POS\_COLOR\_TEX);

// vertices

glVertexAttribPointer(GLProgram::VERTEX\_ATTRIB\_POSITION, 3, GL\_FLOAT, GL\_FALSE, kQuadSize, (GLvoid\*) offsetof(V3F\_C4B\_T2F, vertices));

// colors

glVertexAttribPointer(GLProgram::VERTEX\_ATTRIB\_COLOR, 4, GL\_UNSIGNED\_BYTE, GL\_TRUE, kQuadSize, (GLvoid\*) offsetof(V3F\_C4B\_T2F, colors));

// tex coords

glVertexAttribPointer(GLProgram::VERTEX\_ATTRIB\_TEX\_COORD, 2, GL\_FLOAT, GL\_FALSE, kQuadSize, (GLvoid\*) offsetof(V3F\_C4B\_T2F, texCoords));

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, \_buffersVBO[1]);

glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(\_indices[0]) \* \_filledIndex, \_indices, GL\_STATIC\_DRAW);

}

//Start drawing vertices in batch

**for(const auto& cmd : \_batchedCommands) // 一次性找完**

{

**auto newMaterialID = cmd->getMaterialID();**

**if(\_lastMaterialID != newMaterialID || newMaterialID == MATERIAL\_ID\_DO\_NOT\_BATCH) // 判断材质是否相同**

{

//Draw quads

if(indexToDraw > 0)

{

glDrawElements(GL\_TRIANGLES, (GLsizei) indexToDraw, GL\_UNSIGNED\_SHORT, (GLvoid\*) (startIndex\*sizeof(\_indices[0])) );

\_drawnBatches++;

\_drawnVertices += indexToDraw;

startIndex += indexToDraw;

indexToDraw = 0;

}

//Use new material

cmd->useMaterial();

\_lastMaterialID = newMaterialID;

}

indexToDraw += cmd->getIndexCount();

}

//Draw any remaining triangles

if(indexToDraw > 0)

{

glDrawElements(GL\_TRIANGLES, (GLsizei) indexToDraw, GL\_UNSIGNED\_SHORT, (GLvoid\*) (startIndex\*sizeof(\_indices[0])) );

\_drawnBatches++;

\_drawnVertices += indexToDraw;

}

if (Configuration::getInstance()->supportsShareableVAO())

{

//Unbind VAO

GL::bindVAO(0);

}

else

{

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

}

\_batchedCommands.clear();

\_filledVertex = 0;

\_filledIndex = 0;

}

一个例子（我们在一个Layer添加三个精灵）



此时rendercall为1，在上面的例子中，Scale9Sprite与Sprite默认的globalZ均为0，其命令也均为三角形命令

**添加下面两句代码：**

progressBackground->setGlobalZOrder(-0.5f);

tempSprite->setGlobalZOrder(1.0f);



此时rendercall变为3

**关于材质（实际上就是一个简单的hash值，QuadCommand的生成与TrianglesCommand的算法一样）：**

**void TrianglesCommand::generateMaterialID()**

{

if(\_glProgramState->getUniformCount() > 0)

{

\_materialID = Renderer::MATERIAL\_ID\_DO\_NOT\_BATCH;

}

else

{

int glProgram = (int)\_glProgramState->getGLProgram()->getProgram();

int intArray[4] = { glProgram, (int)\_textureID, (int)\_blendType.src, (int)\_blendType.dst};

\_materialID = XXH32((const void\*)intArray, sizeof(intArray), 0);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Simple Hash Functions

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

unsigned int XXH32 (const void\* input, int len, unsigned int seed);