Two-LLM Pipeline Experiment

mshell Ecosystem · Art2Dec SoftLab · February 21, 2026

Overview

This document describes a live experiment with the ***mshell Ecosystem*** *— an AI-powered polyglot shell that supports C, C++, Rust, Go, Python, Lua, Bash and native mshell scripting. The experiment demonstrates a* ***two-model LLM pipeline*** *where two AI models collaborate to generate, review, improve, and execute a C/OpenGL 3D visualization program — all driven from a single Markdown document.*

The pipeline follows three stages:

```
Model 1  →  generates C code
Model 2  →  reviews and improves the code
Model 1  →  final review + exec mode (mshell compiles and runs)
```

The Pipeline Document

Below is the complete `.md` file used to drive the experiment. Each block is explained in the Step-by-Step Log section below.

```
### Motion Visualization Pipeline
### Two-model LLM pipeline: generate → review/improve → execute

<!--@1 >c_code
Write a C program that visualizes a solar system simulation:
a central sun and two planets orbiting around it at different speeds and radii.
Use colored spheres drawn manually with triangle strips. Window size 800x600.
Planets must orbit forever without stopping.
Requirements:
- Use GLFW for window creation and event loop
- Use GLEW for OpenGL extension loading
- Use OpenGL 3D perspective (glm-style math with manual matrices is fine)
- Use gluPerspective and gluLookAt for camera (include <GL/glu.h>)
- Draw spheres using a manual sphere drawing function (latitude/longitude loops with GL_TRIANGLE_STRIP)
- Use glRotatef and glTranslatef for orbits
- Use glfwSetTime or a delta-time loop for animation
```

~~~markdown
- Required includes: #include <GL/glew.h>, #include <GLFW/glfw3.h>, #include
<GL/gl.h>, #include <math.h>
- Required compile flags: -lglfw -lGLEW -lGL -lm
Write only the C code, no explanation, no markdown fences.
-->

```bash
echo "=== Model 1 generated the following C code ==="
cat $MSH_VAR_c_code
echo "=== End of generated code ==="
```

<!--@2 <c_code >c_code_improved
You are a senior C/OpenGL developer. Review the C code provided.
MSH_VAR_c_code environment variable contains a FILE PATH — read that file to
get the code.
Check for: missing includes, incorrect OpenGL/GLFW/GLEW calls, animation
bugs, compilation issues.
Improve the code if needed: correct GLFW init/loop, proper GLEW init after
context creation, lighting.
All includes must be present: #include <GL/glew.h>, #include <GLFW/glfw3.h>,
#include <GL/gl.h>, #include <math.h>
GLEW must be initialized after glfwMakeContextCurrent.
Return only the improved C code, no explanation, no markdown fences.
-->

```bash
echo "=== Model 2 reviewed and improved the code ==="
cat $MSH_VAR_c_code_improved
echo "=== End of improved code ==="
```

<!--@1x <c_code_improved
MSH_VAR_c_code_improved environment variable contains a FILE PATH — read that
file to get the C code.
Do a final review: check includes, GLFW/GLEW/OpenGL calls, animation loop,
compilation readiness.
If everything looks correct return the code as-is.
If you find any issues fix them.
Return the final C code in a c markdown block:
```c
... code here ...
```
-->
~~~

Step-by-Step Log

Step 1 — Model 1 generates the initial C code

Directive: `<!--@1 >c_code -->`

Model 1 receives a detailed prompt specifying the full tech stack (GLFW + GLEW + OpenGL legacy pipeline) and produces a complete C program. The output is saved to the session variable `c_code`.

Key decisions made by the model: - GLFW 2.1 context with `glfwWindowHint` - Sphere drawn manually via `GL_TRIANGLE_STRIP` with latitude/longitude loops - Two planets animated using `glRotatef(glfwGetTime() * speed, 0, 1, 0)` — orbits never stop - `glewInit()` correctly placed after `glfwMakeContextCurrent()` - Projection set via `gluPerspective` + `gluLookAt` camera at `(0, 50, 100)`

A Bash block then prints the generated code to the terminal for inspection:

```
echo "=== Model 1 generated the following C code ==="
cat $MSH_VAR_c_code
echo "=== End of generated code ==="
```

Step 2 — Model 2 reviews and improves the code

Directive: `<!--@2 <c_code >c_code_improved -->`

Model 2 acts as a **senior C/OpenGL developer**. It reads the code from the file path in `MSH_VAR_c_code`, audits it, and returns an improved version saved to `c_code_improved`.

Improvements made by Model 2: - Added `#ifndef M_PI` guard — portable across all compilers - Fixed sphere loop boundary: `i < SPHERE_SLICES` → `i <= SPHERE_SLICES` for correct vertex count - Added full OpenGL lighting setup: `GL_LIGHT0` with ambient, diffuse and specular components - Added `GL_COLOR_MATERIAL` + `glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE)` so `glColor3f` drives material color correctly - Added sun emission material so the sun glows independently of lighting

Another Bash block prints the improved version:

```
echo "=== Model 2 reviewed and improved the code ==="
cat $MSH_VAR_c_code_improved
echo "=== End of improved code ==="
```

Step 3 — Model 1 final review + exec mode

Directive: `<!--@1x <c_code_improved -->`

Model 1 receives the improved code for a **final pass**, then returns it wrapped in a `` ```c ``` `` block. mshell's exec mode (x) intercepts the response, extracts the code from the fenced block, hands it to the **mshell Validator**, which:

1. Detects includes (`GL/glew.h`, `GLFW/glfw3.h`, `GL/gl.h`, `GL/glu.h`) and resolves library flags automatically via the active **OpenGL 3D** profile
2. Runs the compiler:

```
[DEBUG compile_c] libs: [-lm -L/usr/lib64 -lGLEW -lGL -lX11 -lGLU -lglfw]
[DEBUG compile_c] cmd: [gcc -Wall -Wextra -std=c11 -o
/tmp/mshell_validate_12864.out
    /tmp/mshell_validate_12864.c -lm -L/usr/lib64 -lGLEW -lGL -lX11 -lGLU -
lglfw 2>&1]
```

3. Compilation succeeds — the binary launches and the **Solar System** window opens.

Final changes made by Model 1 in this pass: - Switched sphere rendering from `GL_TRIANGLE_STRIP` to `GL_QUAD_STRIP` — cleaner quad mesh, no vertex duplication artifacts - Added specular material for planets: `GL_SHININESS = 50.0` for a shiny surface effect - Removed specular from the sun — a light source does not reflect light - Confirmed correct light position update inside `render()` after `glLoadIdentity()`

Code: Model 1 Output (Initial Generation)

The raw code produced by Model 1 before any review:

```c
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <stdio.h>

#define WIDTH 800
#define HEIGHT 600
#define SPHERE_SLICES 32
#define SPHERE_STACKS 32

GLFWwindow* window;

void drawSphere(float radius) {
    float theta, phi;
    float x, y, z;
    float dx, dy;

    glBegin(GL_TRIANGLE_STRIP);
    for (int i = 0; i < SPHERE_SLICES; i++) {
        theta = (2.0 * M_PI * i) / (SPHERE_SLICES - 1);
```

```
        for (int j = 0; j <= SPHERE_STACKS; j++) {
            phi = (M_PI * j) / SPHERE_STACKS;

            x = radius * sin(phi) * cos(theta);
            y = radius * cos(phi);
            z = radius * sin(phi) * sin(theta);
            glVertex3f(x, y, z);

            x = radius * sin(phi) * cos(theta + 2.0 * M_PI / (SPHERE_SLICES -
1));
            y = radius * cos(phi);
            z = radius * sin(phi) * sin(theta + 2.0 * M_PI / (SPHERE_SLICES -
1));
            glVertex3f(x, y, z);
        }
    }
    glEnd();
}

void render() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 50.0, 100.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    double time = glfwGetTime();

    // Draw Sun
    glColor3f(1.0, 1.0, 0.0);
    drawSphere(5.0);

    // Draw Planet 1 (inner)
    glPushMatrix();
    glRotatef(time * 50.0, 0.0, 1.0, 0.0);
    glTranslatef(20.0, 0.0, 0.0);
    glColor3f(0.5, 0.7, 1.0);
    drawSphere(2.0);
    glPopMatrix();

    // Draw Planet 2 (outer)
    glPushMatrix();
    glRotatef(time * 20.0, 0.0, 1.0, 0.0);
    glTranslatef(40.0, 0.0, 0.0);
    glColor3f(1.0, 0.5, 0.5);
    drawSphere(3.0);
    glPopMatrix();

    glutSwapBuffers();
```

```
}

void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (double)width / (double)height, 1.0, 500.0);
    glMatrixMode(GL_MODELVIEW);
}

int main() {
    if (!glfwInit()) {
        return -1;
    }

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Solar System", NULL, NULL);
    if (!window) {
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    if (glewInit() != GLEW_OK) {
        return -1;
    }

    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (double)WIDTH / (double)HEIGHT, 1.0, 500.0);

    while (!glfwWindowShouldClose(window)) {
        render();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();
    return 0;
}
```

Code: Model 2 Output (After Review & Improvement)

Model 2 reviewed the above and returned this improved version:

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <stdio.h>

#define WIDTH 800
#define HEIGHT 600
#define SPHERE_SLICES 32
#define SPHERE_STACKS 32

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

GLFWwindow* window;

void drawSphere(float radius) {
    float theta, phi;
    float x, y, z;

    glBegin(GL_TRIANGLE_STRIP);
    for (int i = 0; i <= SPHERE_SLICES; i++) {
        theta = (2.0 * M_PI * i) / SPHERE_SLICES;
        for (int j = 0; j <= SPHERE_STACKS; j++) {
            phi = (M_PI * j) / SPHERE_STACKS;

            x = radius * sin(phi) * cos(theta);
            y = radius * cos(phi);
            z = radius * sin(phi) * sin(theta);
            glVertex3f(x, y, z);

            x = radius * sin(phi) * cos(theta + 2.0 * M_PI / SPHERE_SLICES);
            y = radius * cos(phi);
            z = radius * sin(phi) * sin(theta + 2.0 * M_PI / SPHERE_SLICES);
            glVertex3f(x, y, z);
        }
    }
    glEnd();
}

void render() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 50.0, 100.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    double time = glfwGetTime();

    // Set light position at sun's location
    GLfloat light_pos[] = {0.0, 0.0, 0.0, 1.0};
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);

    // Draw Sun
    glColor3f(1.0, 1.0, 0.0);
    GLfloat sun_emission[] = {0.5, 0.5, 0.0, 1.0};
    glMaterialfv(GL_FRONT, GL_EMISSION, sun_emission);
    drawSphere(5.0);

    // Reset emission for planets
    GLfloat no_emission[] = {0.0, 0.0, 0.0, 1.0};
    glMaterialfv(GL_FRONT, GL_EMISSION, no_emission);

    // Draw Planet 1 (inner)
    glPushMatrix();
    glRotatef(time * 50.0, 0.0, 1.0, 0.0);
    glTranslatef(20.0, 0.0, 0.0);
    glColor3f(0.5, 0.7, 1.0);
    drawSphere(2.0);
    glPopMatrix();

    // Draw Planet 2 (outer)
    glPushMatrix();
    glRotatef(time * 20.0, 0.0, 1.0, 0.0);
    glTranslatef(40.0, 0.0, 0.0);
    glColor3f(1.0, 0.5, 0.5);
    drawSphere(3.0);
    glPopMatrix();
}

void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (double)width / (double)height, 1.0, 500.0);
    glMatrixMode(GL_MODELVIEW);
}

int main() {
    if (!glfwInit()) {
        return -1;
    }
```

```
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Solar System", NULL, NULL);
    if (!window) {
        glfwTerminate();
        return -1;
    }

    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    if (glewInit() != GLEW_OK) {
        return -1;
    }

    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (double)WIDTH / (double)HEIGHT, 1.0, 500.0);

    // Enable lighting
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    GLfloat light_ambient[]  = {0.2f, 0.2f, 0.2f, 1.0f};
    GLfloat light_diffuse[]  = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    glMatrixMode(GL_MODELVIEW);

    while (!glfwWindowShouldClose(window)) {
        render();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();
    return 0;
}
```

Code: Final Version (Model 1 exec pass)

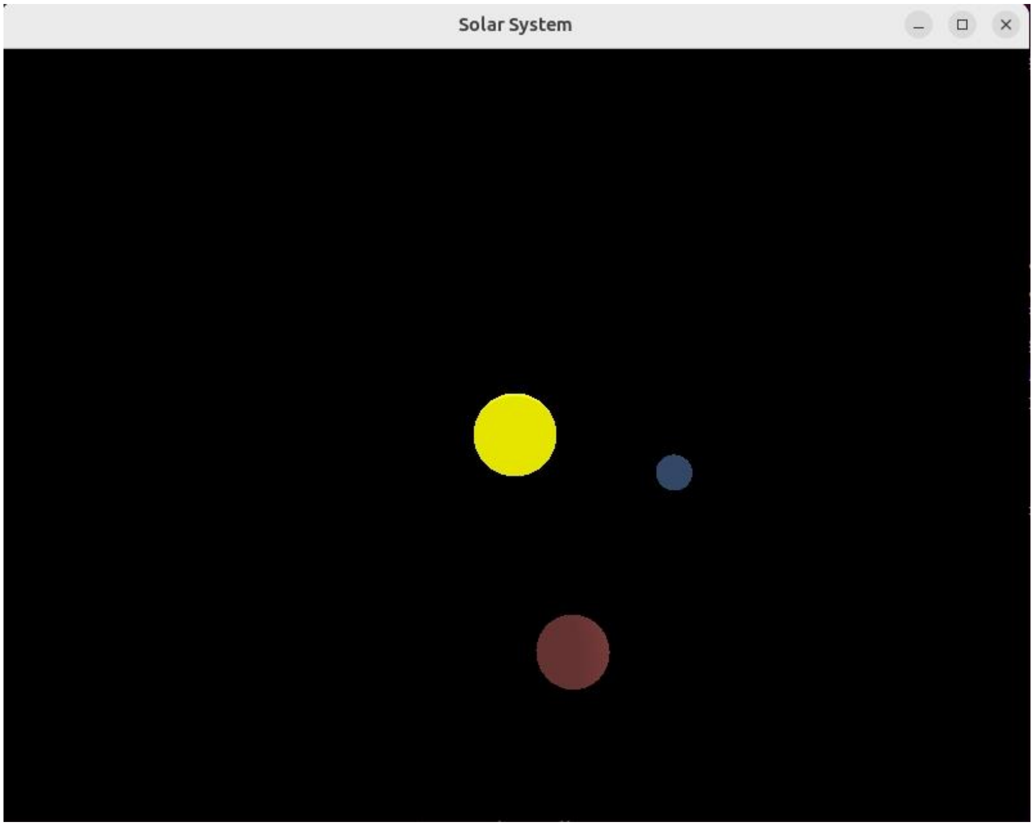
The code that was actually compiled and executed by mshell:

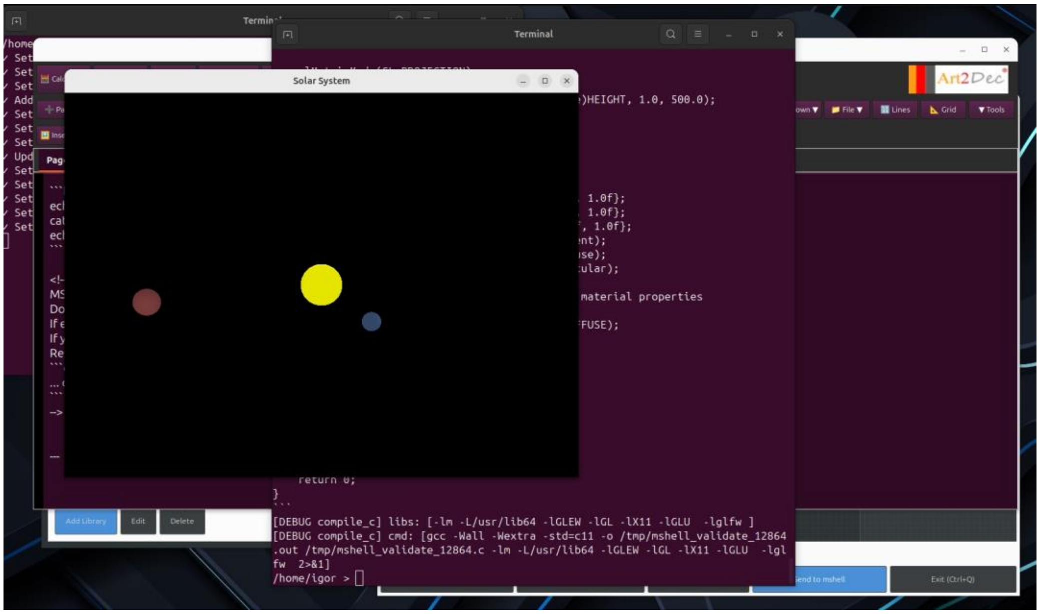
```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <math.h>
#include <stdio.h>

#define WIDTH 800
#define HEIGHT 600
#define SPHERE_SLICES 32
#define SPHERE_STACKS 32

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

GLFWwindow* window;

void drawSphere(float radius) {
    float theta1, theta2;
    float ex, ey, ez;

    glBegin(GL_QUAD_STRIP);
    for (int i = 0; i < SPHERE_SLICES; i++) {
        theta1 = (2.0 * M_PI * i)       / SPHERE_SLICES;
        theta2 = (2.0 * M_PI * (i + 1)) / SPHERE_SLICES;

        for (int j = 0; j <= SPHERE_STACKS; j++) {
            float theta3 = (M_PI * j) / SPHERE_STACKS;

            ex = sin(theta3) * cos(theta1);
            ey = cos(theta3);
            ez = sin(theta3) * sin(theta1);
            glVertex3f(ex * radius, ey * radius, ez * radius);

            ex = sin(theta3) * cos(theta2);
            ey = cos(theta3);
            ez = sin(theta3) * sin(theta2);
            glVertex3f(ex * radius, ey * radius, ez * radius);
        }
    }
    glEnd();
}
```

```
void render() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 50.0, 100.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    double time = glfwGetTime();

    // Light at sun position
    GLfloat light_pos[] = {0.0, 0.0, 0.0, 1.0};
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);

    // Sun — emissive, no specular
    glColor3f(1.0, 1.0, 0.0);
    GLfloat sun_emission[] = {0.5, 0.5, 0.0, 1.0};
    GLfloat sun_specular[]  = {0.0, 0.0, 0.0, 1.0};
    glMaterialfv(GL_FRONT, GL_EMISSION, sun_emission);
    glMaterialfv(GL_FRONT, GL_SPECULAR, sun_specular);
    drawSphere(5.0);

    // Reset for planets — specular highlight enabled
    GLfloat no_emission[]      = {0.0, 0.0, 0.0, 1.0};
    GLfloat planet_specular[]  = {1.0, 1.0, 1.0, 1.0};
    glMaterialfv(GL_FRONT, GL_EMISSION, no_emission);
    glMaterialfv(GL_FRONT, GL_SPECULAR, planet_specular);
    glMaterialf (GL_FRONT, GL_SHININESS, 50.0);

    // Planet 1 — inner, faster
    glPushMatrix();
    glRotatef(time * 50.0, 0.0, 1.0, 0.0);
    glTranslatef(20.0, 0.0, 0.0);
    glColor3f(0.5, 0.7, 1.0);
    drawSphere(2.0);
    glPopMatrix();

    // Planet 2 — outer, slower
    glPushMatrix();
    glRotatef(time * 20.0, 0.0, 1.0, 0.0);
    glTranslatef(40.0, 0.0, 0.0);
    glColor3f(1.0, 0.5, 0.5);
    drawSphere(3.0);
    glPopMatrix();
}

void framebuffer_size_callback(GLFWwindow* window, int width, int height) {
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();
    gluPerspective(45.0, (double)width / (double)height, 1.0, 500.0);
    glMatrixMode(GL_MODELVIEW);
}

int main() {
    if (!glfwInit()) return -1;

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Solar System", NULL, NULL);
    if (!window) { glfwTerminate(); return -1; }

    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    if (glewInit() != GLEW_OK) return -1;

    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (double)WIDTH / (double)HEIGHT, 1.0, 500.0);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    GLfloat light_ambient[]  = {0.2f, 0.2f, 0.2f, 1.0f};
    GLfloat light_diffuse[]  = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    glMatrixMode(GL_MODELVIEW);

    while (!glfwWindowShouldClose(window)) {
        render();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();
```

```
    return 0;
}
```





```
    return 0;
}
```
[DEBUG compile_c] libs: [-lm -L/usr/lib64 -lGLEW -lGL -lX11 -lGLU -lglfw]
[DEBUG compile_c] cmd: [gcc -Wall -Wextra -std=c11 -o /tmp/mshell_validate_12864
.out /tmp/mshell_validate_12864.c -lm -L/usr/lib64 -lGLEW -lGL -lX11 -lGLU -lgl
fw 2>&1]
/home/igor >
```
```

Final Result

The Solar System window opened successfully showing:

- **Yellow sun** at the center with emission glow
- **Blue planet** orbiting at radius 20, speed factor 50
- **Red planet** orbiting at radius 40, speed factor 20
- Black background, 3D perspective camera at `(0, 50, 100)`, depth test enabled
- Animation runs forever — no energy decay, pure time-based rotation

---

Key Takeaways

1. Models genuinely complement each other

Model 2 added a full lighting system that was completely absent in the original. Model 1 in the final pass restructured the sphere geometry, switching from `GL_TRIANGLE_STRIP` to `GL_QUAD_STRIP`. This was not simple proofreading — each pass added something architecturally meaningful. The conclusion is that a multi-model pipeline has real value even when both models are of similar capability class. Splitting responsibilities — generate, review, finalize — produces better results than a single model doing everything in one shot.

2. The “fresh eyes” effect is real

When Model 1 received back the code that had already been processed by Model 2, it found and fixed a vertex ordering problem in `drawSphere` that it had originally introduced itself. This mirrors a well-known pattern in human code review: a developer is less critical of their own code than of someone else’s. Running the same model on code it did not just generate appears to trigger a more analytical mode. This suggests that even a single-model pipeline benefits from a review pass with a deliberately different prompt framing.

A concrete example from this experiment: in the final pass, Model 1 made a physically correct decision to remove specular highlighting from the sun while keeping it on the planets. The reasoning is sound — the sun is a light source, not a reflective surface, so it should not have a specular component responding to itself. The planets, being illuminated objects, benefit from specular highlights to convey surface shininess. This is a subtle distinction that requires understanding the scene’s lighting model, not just the syntax of the OpenGL calls. The model arrived at this decision only when reviewing the code from the outside — it did not make this distinction during initial generation.

3. Prompt specification is more important than it might seem

The quality and specificity of the initial prompt directly determines what the model can realistically produce. Vague requirements lead to code that compiles in theory but not on the actual target system. Explicit constraints — which libraries are available, what includes to use, what compile flags are expected — give the model enough context to generate code that works in the specific environment rather than in a generic one. The more precisely the prompt describes the target, the less correction work the subsequent review passes need to do.

4. A slow model in the pipeline slows down the whole chain

If one model in the sequence is significantly slower than the others, the entire pipeline works at that model's pace. This is important to consider when designing multi-step pipelines: place slower or heavier models only at stages where their deeper reasoning genuinely adds value — such as a final review or an architectural decision — and use faster models for generation and lightweight checks. Matching model capability to task complexity at each stage is key to a responsive pipeline.

5. The Validator as a smart abstraction layer

The mshell Validator's ability to auto-detect required libraries from `#include` directives is a well-designed abstraction between the LLM and the actual build environment. The model writes standard headers, and the Validator resolves the correct linker flags for the local system automatically. This means the LLM does not need to know the exact library paths or system configuration — it only needs to produce correct C code. This separation of concerns makes the pipeline portable: the same prompt and the same generated code can compile correctly across different Linux distributions, library versions, and directory layouts without any changes to the pipeline document itself. It is an architectural pattern worth extending further — for example, auto-detecting missing system packages and suggesting installation commands when a header is present in the code but the library is not installed.

Summary

The experiment demonstrates that mshell's Interlang + LLM pipeline is a viable approach to automated, multi-stage code generation. The combination of session variables for inter-model data exchange, exec mode for seamless compilation, and the Validator's library resolution creates a workflow where the human role shifts from writing and debugging code to designing prompts and orchestrating model collaboration. The final working OpenGL application was produced entirely through this pipeline without any manual code editing.

---

mshell Ecosystem · Art2Dec SoftLab · igor.lukyanov@appservgrid.com
https://www.appservgrid.com/paw92
