Technische Standards und Designvorgaben für die GDL-Programmierung von ArchiCAD®-Bibliothekselementen bei b-prisma

Einführung

Diese Dokumentation enthält Technische Standards und Designvorgaben, welche bei der GDL-Programmierung von ArchiCAD-Bibliothekselementen (GDL-Objekten), welche bei b-prisma erstellt werden, einzuhalten sind.

Außerdem werden einige Regeln und Empfehlungen beschrieben, die nicht zwingend einzuhalten sind, aber empfohlen werden. Sämtliche Standards, Regeln, Empfehlungen, Vorgaben, die NICHT verbindlich sind, sondern nur empfehlenden Charakter haben, sind durch Kursivschrift gekennzeichnet.

Diese Dokumentation wurde erstellt, um einen einheitliche Standard beim Scripting zu gewährleisten, damit alle beteiligten GDL-Programmierer und andere Projektbeteiligten immer auf den selben Standard zugreifen können und sich sofort zurechtfinden.

Im übrigen wird empfohlen, sich an die Graphisoft-Dokumenatationen „Library Design“, „Library Specification“, „GDL-Style-Guide“ und GDL Advanced Technical Standards“ zu halten, soweit Sie nichts anders sagen, als die b-prisma-Standards.

Namensgebung

Im Laufe der Jahre hat sich eine Namensgebung bei b-prisma etabliert, die sich zum Teil an Graphisoft-Vorgaben hält, zum Teil aber bewusst davon abweicht. Die Art der Namensgebung und Groß/Kleinschreibung verfolgt den Zweck, dass ein Maximum an Übersichtlichkeit und Verständnis der Scripte und Parameter- und Variablennamen gegeben ist.

GRUNDSÄTZLICHES

- sämtliche Befehle und Globalen Variablen werden komplett in Großbuchstaben geschrieben, damit sie sich von Parametern und Variablen unterscheiden
- sämtlich Parameternamen und Variablennamen werden entweder gemischt klein/GROSS geschrieben oder klein; komplette Großschreibung ist nicht erlaubt

PARAMETER

- Alle Parameter fangen mit einem kleinen Bezeichner, der den Parametertyp kennzeichnet, an gefolgt von einem Unterstrich, gegliedert von allgemein zu spezifisch.
- der 2. Teil beginnt mit einem Großbuchstaben und beschreibt die Funktion
- der 3. Teil beginnt mit einem Großbuchstaben und beschreibt das Bauteil
- Beispiel: `str_TypFenster (nicht FensterTyp)`, erlaubt ist auch `str_Typ_Fenster`

PARAMETERTYPEN

Zur Unterscheidung der Parametertypen im Script erhalten diese folgend aufgeführten Präfixe gefolgt von Unterstrich und Bezeichnung wie oben.

- Länge: x_Parameter, y_Parameter, z_Parameter
- Winkel: ang_Parameter
- Gleitkommazahl: real_Parameter

- Ganzzahl: int_Parameter
- Boolescher Wert: b_Parameter
- String: str_Parameter
- Schraffur: fill_Parameter
- Material: mat_Parameter
- Stift: pen_Parameter
- Linientyp: line_Parameter

VARIABLEN

- alle im Script definierten Variablen fangen mit einem Unterstrich an
- der erste Teil beginnt klein und beschreibt die Funktion, der zweite Teil beginnt mit einem Großbuchstaben und beschreibt das Bauteil
- Beispiel: _langProfil
- Abweichungen: eps = 0.0001 und htsp = 0 (Vergleichsvariable und Hotspothochzähler)
- Variablen, die nur in einer Subroutine vorkommen, beginnen mit 2 Unterstrichen, z.B.: __iTuerTyp; *alternativ ist auch erlaubt ~typTuer,* erlaubt ist auch _lang_Profil
- Schleifenzähler: als Schleifenzähler kommen NUR Kleinbuchstaben ohne Zahlen und Unterstriche in Frage; man startet immer mit i, gefolgt von j, k, m und n. l sollte man vermeiden, da es schlecht von I und 1 zu unterscheiden ist. Bei Schleifen in Subroutinen sind grundsätzlich Doppelbuchstaben wie ii, jj, kk zu verwenden oder i_1, i_2, etc.

Stile

Folgende Zeichenstile sind ebenfalls einzuhalten. Leerzeichen bedeuten zwar ein gewissen Mehraufwand beim Scripten, erhöhen jedoch die Scriptübersichtlichkeit und damit die Lesbarkeit erheblich. Gleiches gilt für Abschnittsbegrenzungen.

- Vor und nach **=** steht immer ein Leerzeichen, sowie bei Rechenoperanden
- Falsch: `IF i=1 THEN h[n + 6]=h+5 | j = - x | not (x)`
- Richtig: `IF i = 1 THEN h[n+6] = h + 5 | j = -x | not(x) | s = MIN (a, 25 * b, c)`
- Keine Leerzeichen sind erlaubt vor und hinter den folgenden Operatoren:
  Indizierung von Arrays: `array[25]`
  logisches "not": `not(x)`
  unäres minus, unäres plus: `-x, +x`

- Codeabschnitte und Abschnittsbegrenzer bitte wie folgt verwenden: Überschrift in Bindestrichzeilen eingefasst, Leerzeilen über und unter Bindestrichzeilen, Abschnittsbegrenzungstext in G R O S S B U C H S T A B E N mit Leerzeichen dazwischen, was die Übersichtlichkeit weiter erhöht, dann Name des Codeblocks mit >> am Anfang und Name des Codeblocks mit << am Ende:

```
! ---------------------------------------------------------------------- !
! A B S C H N I T T S B E G R E N Z U N G
! ---------------------------------------------------------------------- !
```

```
! ------ C O D E B L O C K   0 0 1 --- >>

    !Hier steht der Codeblock!

! << --- C O D E B L O C K   0 0 1 ---
```

SCRIPTÜBERSCHRIFTEN

Jedes Script soll eine Überschrift erhalten, damit es in der XML-Version des Objektes besser unterschieden werden kann.

```
! ------------------------------------------------------------------- !
! -------- M A S T E R  -  S c r i p t ------------------------------- !
! ------------------------------------------------------------------- !
```

END-Zeilen sind wie folgt anzulegen:

```
! ---------------------------- E N D ---------------------------------- !
! ---------------------------- E N D ---------------------------------- !

END !      END    !      END    !      END    !      END    !      END    !      END

! ---------------------------- E N D ---------------------------------- !
! ---------------------------- E N D ---------------------------------- !
```

Subroutinen sind durch Doppel-Bindestrichzeilen voneinander zu trennen. Der Name der Subroutine kann eine Zahl sein oder ein Text in Anführungszeichen. Dahinter ist immer die Funktion der Subroutine anzugeben. Bei komplexeren Subroutinen empfiehlt es sich, eine kurze Beschreibung, sowie die übergebenden und zurückgegebene Variablen aufzuführen

```
! ------------------------------------------------------------------- !
! ------------------------------------------------------------------- !
! Erzeugt ein 3D-Türband, der bewegliche Teil folgt in Subroutine 101
! übergebene Parameter
!   _durchmesserBand
!   -materialBand
! Rückgabewerte
!   keine
! ------------------------------------------------------------------- !

100:                              ! -------- T Ü R B A N D --------- !

    ! Code der Subroutine ist eingerückt

RETURN

! ------------------------------------------------------------------- !
! ------------------------------------------------------------------- !
```

EINRÜCKUNGEN (TAB-SPRÜNGE)

Bei Kontrollfluss-Anweisungen, Schleifen, sind Tabsprünge, ggf. gestaffelt, zu verwenden.

Beispiele:

```
FOR i = 1 TO anz
    ADDx (i - 1) * A
    ! Codeblock
    DEL 1
NEXT i

IF strTypTuer = „Schwingtür“ THEN
    ! Codeblock
ENDIF

ADDx 1
ROTy 44
! Codeblock
DEL 1
```

HALBFERTIGE ARBEITEN (TODOS)

Bei nicht fertigen Codebereichen soll immer das Schlüsselwort `TODO` als Erinnerung gesetzt werden. So ist es einfacher, die Stelle wiederzufnden, z.B.:

```
ADDx 1     ! TODO: temporäre Verschiebung; später wieder aufheben
! Codebereich
DEL 1
```

MARKIEREN VON ZUSAMMENHÄNGENDEN BEFEHLEN

Bei längeren Scriptbereichen sollen zusammenhängende Befehle entsprechend kommentiert werden, z.B. bei Koordinatentransformationen und Bedingungs-Abläufen, z.B.:

```
ADDx _verschiebungRahmen       ! ko-trafo 01
If Bedingung_1 THEN
    IF Bedingung_2 THEN
           ! Größerer Codebereich
    ENDIF                      ! Bedingung_2
ENDIF                          ! Bedingung_1
DEL 1                          ! ko-trafo 01
```

Parameter-Script

FESTSETZUNGEN VON PARAMETERN

Will man den Wert eines Parameters unter Verwendung des Befehls `PARAMETERS` im Parameter-Script ändern, muss eine ähnliche Festsetzung im Master-Script erfolgen. Dies sorgt für eine korrekte Anzeige des Objektes für Fälle, wenn das Parameter-Script nicht vom System durchlaufen wird. z.B.:

```
! Parameter-Script
if Bedingung_1 then
    str_TypFenster = „Holzfenster“
    PARAMETERS str_TypFenster = str_TypFenster
endif
! Master-Script
if Bedingung_1 then str_TypFenster = „Holzfenster“
```

3D-Script

3D-Darstellung: unbedingt alles im 3D-Kontext weglassen, was nicht sichtbar ist, aber eine hohe Polygonanzahl besitzt und somit Rechenleistung fordert.

Es gibt bestimmte Elemente, die nur im Schnitt sinnvoll sind.

Fehlerkontrolle

In viel verwendeten Bibliothekselementen sollten möglichst umfassende Fehlerkontrollen eingebaut werden, welche verhindern, dass bei Eingabe ungültiger Parameterwerte durch den Anwender keine Fehlermeldungen auftauchen oder sogar das Objekt nicht mehr angezeigt wird:

Einige Beispiele für Fehlerkontrollen:

```
! --- PEN darf nicht kleiner als 1 in 2D sein
IF post_fill_pen < 1 THEN post_fill_pen = 1

! --- prüfe, ob die "Misch-Schraffur 50 %" existiert, dann ist rrr = 1
fill50 = IND(FILL,"Misch-Schraffur 50 %")
framefill = fill50
rrr = REQUEST("Name_of_Fill", framefill, fillName)
```

HOTSPOTS

Sämtliche Hotspots sind mit einer ID zu versehen , auch die nicht beweglichen, damit bei Vermassungen diese dynamische bleiben; bei Hotspots ohne IDs kann es passieren, dass sich die Bemaßungen bei Änderungen der Objektgröße nicht dynamisch anpassen.

Es ist für jedes Objekt samt Makros und Submakros eine Dokumentation über die Verwendung der Hotspot-IDs anzulegen, damit sichergestellt ist, dass es in unterschiedlichen Skripten nicht identische Hotspot-IDs gibt.

Z.B. Hauptobjekt 3D. 1 - 99, Hauptobjekt 2D: 100 - 199

Makro 3D: 1001 - 1099, Makro 2D: 1100 - 1199

Alternativ kann bei Makros auch die ID per Übergabe durchgezählt und per Returned Parameters ins Hauptobjekt zurückgespielt werden, wo dann weitergezählt wird.

In sämtlichen Skripten ist ausschließlich ein einziger identischer Variablenname für das hochzählen von Hotspots einzusetzen. Diese Variable braucht abweichend nicht mit einem Unterstrich zu beginnen.

z.B.: htsp = 1

HOTSPOT2 A, B, htsp : htsp = htsp + 1

Spezifizierung (aus: Graphisofts Library Specification)

Erste Voraussetzung bei der Erstellung eines GDL-Objektes ist, dass sein Zweck und seine Funktion klar definiert sind, um einen perfekten Überblick über die Funktion des Objektes zu bekommen: den Abstraktionsgrad, die Eigenschaften, 2D- und 3D-Darstellung, sowie die Darstellung in Schnitten und Ansichten. Eine der grundlegenden Fragen lautet: wozu soll das Objekt verwendet werden?

Hieraus werden vom GDL-Entwickler in Abstimmung mit dem Auftraggeber sämtliche Parameter und ihre hierarchische Struktur definiert. Die Logik der Parametrik muss sich in der Anordnung der Paarmeter widerspiegeln; die Parameter sollten in Gruppen angeordnet werden und die Verknüpfungen von Parametern untereinander muss klar definiert werden (und später im Parameter-Scipt ausgearbeitet werden). Um dies sicherzustellen, müssen Verbindungen von Parametern untereinander genauestens analysiert werden.

Vom Auftraggeber müssen Spezifizierungs-Zeichnungen erstellt werden, bevor mit der Erstellung eines neuen Bibliothekselementes begonnen wird. Spezifizierungs-Zeichnungen müssen jede Darstellungsvariante jedes Einzelobjektes - wie folgt beschrieben - enthalten:

- diese müssen in unterschiedlichen Maßstäben gezeichnet sein, falls 2D und 3D ihre Darstellung in Abhängigkeit vom Maßstab ändern sollen

- unabhängige Versionen müssen für Fenster/Türen gezeichnet werden, einmal mit und einmal ohne Zusatzelemente, wie Seitenlichter, Oberlichter, Riegel, Pfosten, Kämpfer, Wandanschlüsse etc.

- diese sollten Grundrisssymbol, Schnitt- und Ansichtssymbole in durchgängig einheitlicher Darstellungsart enthalten

- Sämtliche durch Parameter veränderlichen Werte sind mit einheitlichen Buchstaben (Grundriss, Schnitt, Ansicht) in Maßketten zu kennzeichnen; sämtliche Fixgrößen sind durch Zahlenwerte zu kennzeichnen

- Der Programmierer sollte auch für Fixgrößen später im Script Variablen verwenden, um spätere Änderungen zu vereinfachen. Um unerwartete Probleme zu vermeiden, sollte für sämtliche Attribute (Stift oder Schraffur) ein zusätzlicher oder interner Parameter verwendet werden, den man von einer zur anderen Version leicht ändern kann.

- Die Zeichnungen müssen das Modell in jedem Fall so zeigen, wie es hinterher aussehen soll. Um Missverständnisse zu vermeiden, können keine Skizzen akzeptiert werden, die weniger detailliert sind als das gewünschte Ergebnis.

- Die Spezifizierungs-Zeichnungen müssen **alles** enthalten, das für den **Auftraggeber** wichtig ist. Der Programmierer kann Dinge, die nicht in den Spezifizierungs-Zeichnungen gezeigt werden, auch nicht mit in das Objekt einbauen.

Das Entscheidende ist eine gute Kommunikation zwischen Auftraggeber und Programmierer. Dies ist u.a. auch dafür wichtig, damit nicht schwerwiegende Änderungen in einer fortgeschrittenen Phase der Objekt-Entwicklung notwendig werden.

Prototypen (aus: Graphisofts Library Specification)

Normalerweise ist es unmöglich, eine klare Spezifizierung für eine bisher noch nicht vorhandene Funktionalität im Voraus aufzustellen. Prototypen sind daher eine gute Lösung für Auftraggeber und Programmierer, um die Anforderungen und Spezifizierungen untereinander zu kommunizieren. Um die Anforderungen und deren mögliche Lösungen in einer fortgeschrittenen Form darzustellen, sollten ein Prototyp-Objekt für ein Objekt bzw. für grundlegende Objekttypen einer Bibliothek erzeugt werden.

Prototypen müssen vom Auftraggeber sehr genau geprüft werden. Sie sollten sämtliche Parameter, Optionen und Eigenschaften des fertigen Bibliothekselementes enthalten. Sehr oft ist dies in einer frühen Phase der Entwicklung nicht möglich, so dass Bibliothekselemente so vorzubereiten sind, dass Modifikationen leicht möglich sind. Aus diesem Grund sollte ein Bibliothekselement niemals als die ultimative Lösung für die Anforderungen des Auftraggebers in Betracht gezogen werden.